Constraint Programming for Random Testing of a Trading System

Roberto Castañeda Lozano

Master's thesis. Stockholm, January 28, 2010.
School of Information and Communication Technology
KTH Royal Institute of Technology

Constraint Programming for Random Testing of a Trading System

Roberto Castañeda Lozano

Supervisors

Christian Schulte
(Royal Institute of Technology, Sweden)

Lars Wahlberg
(Cinnober Financial Technology AB, Sweden)

Federico Barber
(Polytechnic University of Valencia, Spain)

Examiner

Christian Schulte

Abstract

Financial markets use complex computer trading systems whose failures can cause serious economic damage, making reliability a major concern. Automated random testing has been shown to be useful in finding defects in these systems, but its inherent test oracle problem (automatic generation of the expected system output) is a drawback that has typically prevented its application on a larger scale.

Two main tasks have been carried out in this thesis as a solution to the test oracle problem. First, an independent model of a real trading system based on constraint programming, a method for solving combinatorial problems, has been created. Then, the model has been integrated as a true test oracle in automated random tests. The test oracle maintains the expected state of an order book throughout a sequence of random trade order actions, and provides the expected output of every auction triggered in the order book by generating a corresponding constraint program that is solved with the aid of a constraint programming system.

Constraint programming has allowed the development of an inexpensive, yet reliable test oracle. In 500 random test cases, the test oracle has detected two system failures. These failures correspond to defects that had been present for several years without being discovered neither by less complete oracles nor by the application of more systematic testing approaches.

The main contributions of this thesis are: (1) empirical evidence of both the suitability of applying constraint programming to solve the test oracle problem and the effectiveness of true test oracles in random testing, and (2) a first attempt, as far as the author is aware, to model a non-theoretical continuous double auction using constraint programming.

Keywords: constraint programming, random testing, trading system, test oracle, continuous double auction.

Acknowledgements

This master's thesis has benefited from the advice, guidance and deep knowledge of my supervisors Christian Schulte and Lars Wahlberg, who are truly experts in their respective fields. I am profoundly grateful to them.

I am indebted to my girlfriend Eleonore Lövgren and my good friend and colleague Carles Tomás Martí for providing valuable feedback on earlier versions of this report.

Finally, I wish to thank my parents and my little brother for their support and love, which I strongly feel despite the long distance that keeps us apart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

A financial market is a system that allows buyers and sellers to trade and exchange items of value of different types. Financial markets have existed since very early in the history of humanity, having their origin in the trading of traditional commodities such as agricultural products. Trading of more abstract instruments, such as equities, futures and options appear in Europe at the beginning of the seventeenth century. Since then, new and more complex instruments such as structured financial products have been progressively added to the financial markets [34].

During the last decades, many major regulatory changes have been introduced as a result of different economic and social factors. Furthermore, the constant growth of computer systems and the development of the Internet are revolutionizing the way in which financial markets operate. The traditional way of trading in which traders meet in a physical space to exchange their items (called *floor trading*) is disappearing. At the same time, computer trading agents are substituting gradually the human ones, because of the better responsiveness and precision that is obtained. These facts are increasing the need for more stable, flexible and efficient computer-based trading systems [34, 7].

As a consequence of this need, many recently created financial software companies are appearing. One of them is Cinnober Financial Technology AB (hereinafter called just Cinnober), a Swedish company founded in 1998 that is specialized in the development of computer trading systems, using as a base their *TRADExpress* system [5]. Cinnober employs currently over 140 people, and has delivered solutions to different customers around the world, such as London Metal Exchange, Chicago Board Options Exchange or Hong Kong Mercantile Exchange.

In this environment, systems that handle the daily exchange of millions of highly heterogeneous trade orders, performed by different market participants over many

different kinds of financial instruments are necessarily complex. Furthermore, the fact that failures can cause serious economic damage makes them critical. Cinnober's *TRADExpress Trading System* is a such computer system developed in Java that comprises millions of lines of code, several layers of abstraction and a fully redundant architecture with several replicated components [5]. Because of both the complexity and criticality of *TRADExpress*, a high importance is given to the testing process.

Cinnober is constantly looking for the most efficient way to find defects in the developed systems. One of the latest explored techniques that has shown to be very effective for this purpose, always in combination with more systematic approaches, is random testing. Its general idea is to test the system using randomly generated data as input and an independent source from which it is possible to decide the correctness of the system's output [12]. The company is currently studying the most accurate way to apply this technique to its systems, in order to improve the reliability in those parts where it is specially hard to anticipate failures by applying more traditional test approaches.

1.2 Previous work at Cinnober

There are some previous master's theses developed at Cinnober that deal with the test environment and the application of random testing techniques. The frameworks available at Cinnober for automated testing of *TRADExpress*, among other systems, are carefully studied and evaluated in [24]. The master's thesis reported in [19] examines in great detail the theory around random testing and proposes a practical study, where a random trading simulation framework is developed together with a basic failure detection system, and the results of different runs are presented and analyzed. This work is refined by [21], which uses a more realistic input data model for the random trading simulations, based on the application of statistical methods to real trading data.

A big part of the work done by [19] and [21] is developed around the order book, a core component of trading systems in general, and of *TRADExpress* in particular, where different trade orders referring to a certain instrument are stored and matched. Due to the huge order type variety that *TRADExpress* allows nowadays, and the complex logic that their process requires, the order book gets a high priority in the Cinnober's test planning.

1.3 Problem statement

Despite the great progress made by the previous work performed at Cinnober, the developed trading simulation system currently lacks a fully satisfactory way of automatically evaluating the output generated by the order book. This problem is commonly known as the *oracle problem* [9]. The failures that can be currently detected in simulation time are rather trivial to observe, such as system crashes or negative prices in trades [19]. Both [19] and [21] agree that this represents an important limitation in their developed simulation system with respect to the capacity of detecting system defects. Consequently, the development of a system that can automatically predict the correct system's output during the trading simulation is suggested as further work in both cases. Since the order book is the main object of the trading simulation, the core of the problem is to develop a system that is able to calculate the expected output of the order book throughout the trading simulation.

The development of a such system following an imperative programming paradigm similar to the one used in *TRADExpress* could become as complex as for the original system. There is a need, therefore, to find a less costly and error prone modeling and implementation technique, which might give up some benefits of the *TRADExpress* approach, such as run-time efficiency, in return.

1.4 Proposed solution

The object of this master's thesis is to study, design, implement and evaluate a prototype solution as a response to the problem defined in section 1.3. More specifically, the following two main basic tasks will be carried out:

1. Design and implement a model for the order book as it is defined in Cinnober's *TRADExpress*.
2. Integrate the implemented model into the existing trading simulation framework, as a way to evaluate the correctness of the order book's output.

A test oracle is a source that determines expected results from the system under test [11]. The aim is to use the order book model as a test oracle in the trading simulations, reusing as much as possible the work performed by [19] and [21] at Cinnober. The existing trading simulation framework will be adapted to introduce the test oracle, and the benefits will be examined by collecting results of different simulation runs and comparing them to the ones obtained in the previous work introduced in section 1.2.

Constraint programming is a declarative programming paradigm where constraints over variables are defined, and a general purpose constraint system is used

to solve them [30]. This paradigm will be applied for designing and implementing the order book model that will serve as a test oracle. Constraint programming contributes to the solution by offering a pure declarative technique to model and implement the test oracle, allowing to perform the modeling task in a notation that is much closer to our understanding and removing a great part of the implementation effort [8].

1.4.1 Goals

This master's thesis contains a large practical component. The work performed can be seen as a practical experiment, where two techniques with very different backgrounds and areas of application such as constraint programming and random testing are combined with the initial purpose of helping to increase the reliability of a trading system. Two main goals are set, analogously to the tasks defined above:

1. Evaluate the suitability of constraint programming as a modeling technique for complex existing systems such as *TRADExpress*.
2. Evaluate the improvement in the testing process caused by the addition of a more complete test oracle to the random trading simulation.

1.4.2 Success criteria

In the ideal case, a single performance measure determines the degree in which a project succeeds with respect to the defined goals. In the case of this master's thesis, the heterogeneity of the combined concepts, and the dual nature of the set goal make it more feasible to define a combination of measures, which can be divided into two main groups according to the goal whose achievement is measured:

1. Suitability of constraint programming as a modeling technique for the trading system's order book:
   - Number of order attributes that are successfully modeled (that is, which make the test oracle calculate the correct system's expected behavior).
   - Number of functional requirements reflected on the order book model.
   - Number of lines of code of the order book model with respect to the number of lines of code of the system's order book.
2. Improvement in the random trading simulation framework and benefits of using a more complete test oracle in the test process:
   - Number of defects found in the system as a result of running the random trading simulation with the order book model acting as a test oracle.

- Number of order attributes that can take random values as input in the simulation and whose effect in the system can be anticipated by the test oracle.

1.5 Related research

I am not aware of the existence of previous work where constraint programming has been used in support of testing a trading system. However, some research effort has been identified in the combination of each pair of areas: random testing applied to trading systems, constraint programming applied to random testing and constraint programming applied to trading systems.

1.5.1 Random testing and trading systems

Because of the high complexity and criticality of current trading systems, it is reasonable to suppose that hard testing efforts are applied to them. However, these efforts are usually carried out by the industry and so far do not seem to have attracted the attention of the research community. Moreover, the generated documentation, if any, is generally not widely accessible. The identified work is limited therefore to what is cited in section 1.2, together with another master's thesis that deals with the development of random tests for the trading system developed at OMX Technology [12].

1.5.2 Constraint programming and random testing

There is a significant research activity in the application of constraint programming techniques to automatic test data generation. In this application, test input is generated based either on the structure of some piece of code, with the aim of maximizing some coverage criteria [28, 10], or directly on formal specifications [1, 25]. In the second case, expected results can be automatically generated, together with the test input.

Section 1.4 states that constraint programming will be used in this project to construct a system model to be used in random tests. The use of constraint programming as a technique to model existing systems, not necessarily well-defined, for testing purposes does not seem to have been thoroughly explored by the research community, although there exist some remarkable efforts, for example, in the field of hardware testing [2].

1.5.3 Constraint programming and trading systems

The system to be modeled in this project can be seen as a *continuous double auction* (which is the term typically used in the research literature). The study of this problem has attracted much interest from different areas. Several articles present theoretical models formulated using constraint programming [31, 23] or closely related techniques such as integer programming [20].

Unlike the work presented above, this project aims to create a model of an existing trading system, which is used in several financial markets around the world. Aspects like order priority or visibility, which are usual issues in most of the existent trading systems [14, Ch. 2, Appx.], do not seem to have been previously considered in research constraint programming models.

1.6 Report overview

This report can be clearly divided into two main blocks: the first four chapters introduce the problem that the thesis deals with and its proposed goals, as well as the basic knowledge that the reader may need to understand the developed work; the rest of the chapters explain the development of the proposed solution, its evaluation and the drawn conclusions. The most frequent terms related to the main concepts used in this work are compiled in a glossary at the end of the report.

Chapter 2 deals with the foundations of automated random testing of software systems, with a focus on the test design process.

Chapter 3 shows the main principles of constraint programming: how problems can be specified, and how they are typically solved by constraint programming systems.

Chapter 4 includes an overview of Cinnober's TRADExpress system, with a focus on those aspects that are relevant for the aim of this project.

Chapter 5 contains a description of the design process of the order book model and the way it is introduced into the trading simulation framework. The chapter explains and justifies the main implementation decisions.

Chapter 6 shows results related to each goal presented above and discusses in which degree the chosen solution approach solves the presented problem.

Chapter 7 presents the drawn conclusions and proposes possible further work in the area.

Chapter 2

Random testing

2.1 Introduction

The development of a software system is constantly exposed to errors, understood as human actions that produce incorrect results. As a consequence of an error, a software system may contain *defects*, also called *bugs*: flaws that may cause the system to fail to perform its required function. The execution of a software system that contains defects may finally cause deviations from its expected behavior, called *failures* [11].

A desirable goal for all software systems is to reduce their number of defects, so that less failures are experienced when using it. Another important goal is to ensure that a system meets the specified requirements and serves the purpose for which it has been created. These goals are achieved in software development by applying testing: software testing is the process of both detecting defects in a system and determining if it meets the requirements and fits its purpose [11].

Software testing can be roughly classified into functional and non-functional testing. Functional testing analyzes the specification of the system's functionality, while non-functional testing analyzes the system's quality characteristics, such as its performance or its usability. Functional testing is often also referred to as *black-box* testing, because it does not refer to the system's internal structure (even though black-box testing can be also applied to non-functional testing) [11].

Software testing can be classified as well into manual testing (performed manually) and automated testing (performed with the help of a computer system). Because of its flexibility, manual testing can be very effective in finding system failures, but when tests are too time consuming or require high precision, it might be a better option to automate them. In this way, the cost of running several times a test case is reduced, and greater consistency and repeatability can be achieved, which is very important, for example, for regression testing purposes [11].

An automated test case that does not change from one execution to another is considered to be only at a first stage of automation [17]. A more advanced level in test automation includes auto-generation of test cases, and mechanisms to automatically detect system failures. Random testing is a black-box technique that can be found on this level: in random testing, the test cases are selected randomly over a pre-selected domain, and a so-called *oracle* is typically used to automatically detect failures in the system. By randomizing automatic tests, a greater coverage is obtained, and new errors can be found in every new test run [17].

The following sections in this chapter deal with three main aspects that are critical for a successful application of random testing: section 2.2 explains how the input data can be designed and generated, section 2.3 shows how oracles are integrated in the testing model and which types of oracles can be designed, and section 2.4 discusses different aspects related with the management of the output obtained from the application of random testing.

2.2 Input

There are three main characteristics that determine the input in random tests: the domain from which the input is extracted, the distribution that the input data follows, and the size of the input. How these characteristics are designed is a key decision in random testing. A poor design of any of them will lead to poor results, regardless of the quality of the oracle or the output analysis that is performed [18].

2.2.1 Domain

The input domain of a complex system can be considered practically infinite, and the selection of a meaningful subset strongly depends on each system and the testing goals. A common approach is to partition the domain into sub-domains corresponding to different functionalities of the system, defining thus clusters of similar input actions. This makes it possible to assign a different probability or weight to each action, making a so-called *operational profile*, which can be used as a model for the test input domain [13].

2.2.2 Distribution

The distribution for the input data is determined by assigning probabilities to the operational profile, whose structure is determined by the considered domain. Two fundamental approaches can be applied, depending on the testing goals: the operational profile can be chosen so that it models realistic system input, helping thus to asses the reliability of the system (see subsection 2.4.2); or it can be chosen so

that it represents completely unrealistic system input, making it possible to detect low frequency defects whose failures would be hard to cause with more systematic testing [12].

Creating a realistic input model for a complex system is not trivial. Different techniques can be applied in order to create an operational profile that matches the expected system usage. A common one is to apply statistical methods to historical data of the software system in order to create a model that is representable as an operational profile. Operational profiles for newly created functionality require alternative estimation methods. Further detail about how to develop operational profiles can be found in [27].

2.2.3 Size

A complex system usually requires that a test case comprises a sequence of actions, specially if it includes persistent state [13]. The question that immediately arises then is how many actions, or long running time is enough for making a meaningful test case? The answer depends, once again, on the testing goals and the structure of the system:

- If the intention is to make a realistic simulation, the running time should be long enough to reproduce a typical sequence of actions that depend on each other. There is usually no need to reproduce a whole typical system session, if such concept exists in the system under test [27, p. 22]. In this case, the running time could be modeled from analysis on real input data.
- If the intention is instead to deliberately create unrealistic random tests, the length can be chosen in an *ad-hoc* manner. For example, the input size can be determined from experience from previous projects, coverage vs. running length optimization, or even randomly as well.

2.3 Oracles

A test oracle can be generally defined as a source to determine expected output to compare with the actual output of the software under test [11]. In the context of automated testing, a test oracle is an alternate program or mechanism that provides that information [16]. There is a broad range of test oracles, from basic routines that check essential properties of the system to duplicates of the system under test that are able to generate the expected output for all possible inputs. The basic model used in random testing with an oracle is shown in figure 2.1.

A critical success factor in random testing is to have a good mechanism to verify the system output. Without applying oracles in random testing only spectacular

failures can be detected [15]. This strategy can still be convenient in systems where inconsistent or erroneous states frequently lead to system crashes (see [19] for an example).

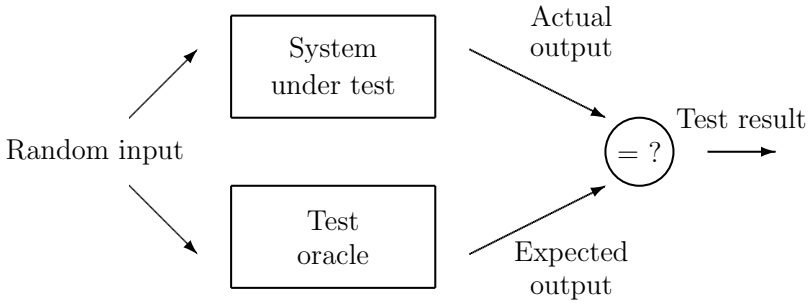


Figure 2.1: Random testing model with an oracle

2.3.1 Characteristics

As mentioned above, many different types of oracles can be used in random testing. Oracles vary widely in attributes such as complexity, completeness or independence from the system under test. Table 2.1 lists the main characteristics that must be considered in the design of an oracle together with a brief description in the form of some questions that each characteristic proposes.

Characteristic	**Description**
Completeness	Which input is covered? Which failures can be detected?
Accuracy	How similar is to the system under test? How independent?
Usability	How is the information given? Does it fit the intended use?
Maintainability	How complex is the oracle? How costly is it to update it?
Performance	How fast does it generate output? How often must be run?
Cost	How much does it cost to make it? How much to execute it?

Table 2.1: Main characteristics of a test oracle *(source: [18])*

As in every software system, many trade-offs must be taken into account in the design of a test oracle. The more complete and accurate an oracle is, the more complex (and therefore less maintainable and more expensive) it will be. A complex oracle is more likely to contain defects and thus lead to false alarms in the comparison between expected and actual output. A complex oracle is generally more sensitive to changes in the system under test, and therefore harder to maintain. If

the test oracle shares components with the system under test, defects can be missed because of both the oracle and the system under test generating the same wrong output due to a defect in the shared component.

2.3.2 Types of oracles

Although different test oracles vary widely in the characteristics discussed in subsection 2.3.1, four main types can be identified: true oracle, consistent oracle, self-referential oracle and heuristic oracle. Not using a test oracle at all, as discussed at the beginning of this section, is sometimes identified as a strategy as well. For a more detailed taxonomy of test oracles, see [15].

The decision of which test oracle type to use depends on many factors, such as the nature of the system under test and the testing goals, the availability of existing oracles, the cost of creating a new oracle and the recognition of useful heuristics. Many times, a combination of different oracle strategies can be the most effective way of making the most of random testing.

True oracle

A true oracle is an independent implementation of the system under test that reproduces the output that is expected from it. The independence can be achieved by, for example, using different algorithms or development platforms, and ensures that the same design defects will not appear in both systems. This type of oracle can be very expensive to implement and maintain, so completeness is not usually achieved. Instead, true oracles are usually applied to test the system for specific input sub-domains.

Consistent oracle

A consistent oracle uses the system output from a previous test run as the expected out for the next one. Therefore, differences in the system output from two consecutive runs are considered as failures by the test oracle. This type of oracle is especially useful for regression testing. Because the correctness of the output from the system under test is not being directly evaluated, a broad range of the input domain can be used. On the other hand, historical defects are likely to remain undetected, as the potential wrong output does not change over the different runs.

Self-referential oracle

A self-referential oracle embeds expected system output within the input data. When using this type of oracle, the random generator seed is usually included in the test input, in order to be able to repeat a specific test run. An independent analysis

is usually run at the end to identify system failures. In this way, the test execution (which is performed first) and the system's output verification are decoupled. A great advantage of this is that post-test analysis can be applied extensively, and it can be always extended without the need of modifying the tests.

Heuristic oracle

A heuristic oracle verifies some characteristics of the system output by exploiting detected patterns in it. Unlike the true oracle, a heuristic oracle cannot decide if a certain system output is correct, but it can be very effective in detecting system failures in cases where there are simple and predictable relationships between input and output. In comparison with the true oracle, a heuristic oracle is simple to develop and maintain (because of its lower complexity), and fast to run. On the other hand, because of its simplicity, it can fail to detect those system failures which break complex patterns and which often respond to the most "interesting" system defects.

2.4 Test output

In order to make it possible to find the defects that exist behind failures detected by random tests, some post-processing is usually needed. But the output of a random test is not only useful to find system failures: some common reliability models can be estimated by applying statistical analysis to it.

2.4.1 Test case reduction

One of the main goals of every kind of software testing is, as discussed at the beginning of this chapter, to find failures in the system under test. When a failure is encountered, the defect that causes it still needs to be found. The process of looking for the defect that causes a certain failure in a system is commonly called *debugging*. Even though debugging falls out of the scope of the testing activities [11], it can be facilitated by simplifying the output of a failed test case.

As discussed in 2.2.3, random test cases for systems with persistent state are usually in the form of sequences of actions, which depending on the system's nature may necessarily be long. When such a test case detects a failure, it can be very hard to find the defect that is causing it, because of the length and complexity of the action sequence. It is usual, therefore, to apply test case reduction to the failed test case in order to find a simpler one that causes the same failure.

A reduction process for a random test case can be very tedious, so automation is very desirable. Delta debugging, for example, is a general automatic technique that

obtains minimal equivalent test cases. The algorithm takes as input the test case and a function that tells whether the test has passed or not. The test case is run then over and over again under changed circumstances until the minimum instance that still causes the system failure is found. Further detail is given in [36, Ch. 13].

2.4.2 Reliability estimation

Software reliability can be defined as the ability of a software system to perform its required functions for a specified period of time or number of operations [11]. Reliability assessment is often included in the list of goals of software testing. By analyzing the output from several random tests applied to a system, various reliability models can be built. A basic measure that is used in many of them is the failure rate, which can be defined as the number of expected failures per time unit and is usually represented by the symbol λ.

A class of models that is widely used in software reliability estimation describes failure processes by non-homogeneous Poisson processes [35, Ch. 4]. By simply calculating the average time between failures from the test output, the Mean Time Between Failures (MTBF) can be obtained, and thus the failure rate $\lambda = \frac{1}{MTBF}$. The number of failures in a period of time is modelled as a variable $X \sim \text{Pois}(\lambda)$. The probability that k failures will occur in a time T can be then calculated in the following way [19]:

$$P\{X = k\} = \frac{(\lambda T)^k}{k!} e^{-\lambda T}$$

The application of reliability models usually require some assumptions, whose validity may depend on the special circumstances of each system. Some of the most usual ones required in times between failure models are [19]:

- Times between failures are independent.
- No new defects are introduced during the defect removal process.
- Each defect has equal probability of exposure.
- The system is immediately repaired after a failure.
- Failure rate is proportional to the number of remaining defects.

The accuracy of a model obtained from random testing output depends not only on the validity of the required assumptions and the statistical confidence of the estimated measures, but also on how realistic the test input is designed.

Chapter 3

Constraint programming

3.1 Introduction

Constraint programming is a programming paradigm used to model and solve problems with a discrete set of solutions. A big part of its power lies in its declarative nature, summarized in this quote:

> Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it. *(Eugene C. Freuder [8])*

In constraint programming, the model of a problem is given by a set of variables with finite domains (so they can take a finite number of different values) and a set of relations between these variables, called constraints, that must hold in every solution [22]. Such a model is called *constraint satisfaction problem* (CSP), or simply *constraint problem*. The solutions to this problem (all possible combination of values for the defined variables for which the given constraints hold) are automatically solved by a constraint programming system. Optionally, a cost function (also called *objective function*) over the variables can be added to the model in order to express preference between several solutions. The solutions become, in this case, the subset of the solutions to the former problem where the cost function is minimized or maximized.

Constraint programming is a particularly useful technique for dealing with combinatorial, typically NP-complete, real life problems. One of its most successful application areas is probably scheduling and planning problems, but there are many others, such as power networks and bioinformatics. Some other areas where constraint programming has recently started to be applied are, for example, formal software verification and optimization of computer programs [30, 33]. In the software testing area, the technique is mainly used to generate test data in automated testing (see subsection 1.5.2).

This chapter aims to introduce constraint programming without assuming previous knowledge of it, working with illustrative examples instead of formal notations. Section 3.2 explains how constraint problems are modeled, and section 3.3 deals with how they can be automatically solved. Last, section 3.4 lists some of the available constraint programming systems together with their basic characteristics.

3.2 Constraint problem modeling

As mentioned above, a constraint satisfaction problem is defined by a set of variables, the finite domain of each variable –typically integers, booleans, floating point numbers or sets of one of these domains–, and a set of constraints. Constraint programming reduces the programming task to the definition of a CSP, which in some cases can be trivial while in some others can be hard, depending mainly on the nature of the problem to model. Currently, there is a great research effort on finding and developing support for new constraints that capture relations between a non-fixed number of variables (called *global* constraints). The application of these constraints gives two main benefits: the programming task becomes easier, and the constraint programing system can be more efficient in solving the specified problem [30, Ch. 6].

Many times, a CSP model leads to several solutions from which only a subset is preferred. This is quite natural if we think in terms of economics, for example (the benefits should be maximized, the costs minimized). In that case, the CSP model can be extended, as explained above, by adding a cost or objective function and stating if it should be maximized or minimized. The cost function, defined over the variables of the model, helps to distinguish the desired solution from those that are not as good as possible.

3.2.1 A classic problem: *Send More Money*

The easiest way to understand how problems are defined in constraint programming is, perhaps, to directly study a case that can be modeled in a natural way as a constraint satisfaction problem. *Send More Money* is a classic mathematical problem published in the early 20th century. The problem defines a combination of words that forms an equation where each letter represents a digit:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

The goal of the problem is to find a different digit for each letter with no leading zeros in the arithmetic expression. The problem, generalized to arbitrary bases, is shown to be NP-complete [6].

3.2.2 CSP formulation

Variables and domain

The variables for this problem are, as one could expect, the different letters that appear in the addition. In this case, all variables have the same (finite) domain. Note that this is not a general requirement for CSP formulations:

$$D, E, M, N, O, R, S, Y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Constraints

Three groups of constraints can be defined for modeling this problem, derived almost directly from its formulation:

Arithmetic expression: the digits represented by the letters must satisfy the algebraic equation:

$$\begin{array}{rrcrcrcrcr} & & & 1000 \times S & + & 100 \times E & + & 10 \times N & + & D \\ & & + & 1000 \times M & + & 100 \times O & + & 10 \times R & + & E \\ \hline = & 10000 \times M & + & 1000 \times O & + & 100 \times N & + & 10 \times E & + & Y \end{array}$$

All letters different: each letter x must represent a different digit:

$$x_i \neq x_j \ , \ \forall i, j : 0 \leq i, j \leq 9, \ i \neq j$$

No leading zeros: leading zeros are not allowed in the numbers represented by each word:

$$S \neq 0, M \neq 0$$

3.2.3 The optimization version: *Send Most Money*

The problem

The *Send Most Money* problem can be defined in a similar way as for *Send More Money*. Each letter represents a different digit, leading zeros are not allowed and the solutions must satisfy the suggested algebraic equation:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & S & T \\ \hline M & O & N & E & Y \end{array}$$

The difference in this case is that only the value combination that satisfy the constraints and maximizes the number formed by the digits represented by the word MONEY is accepted as solution.

The cost function

Having defined the variables, domains and constraints in an analogous way to the *Send Most Money* problem, the only new element that must be introduced is the cost function:

$$maximize \;\; 10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y$$

3.3 Constraint problem solving

As section 3.1 mentions, constraint problems can be automatically solved by computer systems. The main elements used in the computation of the solutions are called *constraint propagation* and *search*, and are respectively discussed in subsections 3.3.1 and 3.3.2. The procedure can be summarized as follows: the possible values for each variable in the model are stored in sets. Values that are forbidden by the problem constraints are discarded from these sets by applying constraint propagation. If a single possible value is obtained for each variable after the propagation, those values are marked as a solution to the problem. If there is some variable with more than one possible value, the problem is decomposed into simpler subproblems and the process is repeated recursively in these.

In order to compute the solutions of a constraint problem with optimization, the same strategy can be used. When the set of solutions is obtained, the one(s) that optimize the cost function can be selected. This trivial model is, however, too inefficient in most cases, because all possible solutions must be found in order to determine which one(s) is the best. Subsection 3.3.2 illustrates a search variant that offers higher performance.

3.3.1 Constraint propagation

Constraint propagation is the kernel operation in constraint programming: without it, constraint programming could be seen as a simple brute-force search. As mentioned above, its mission is to discard the values that cannot appear in a solution,

reducing the number of possible values for the problem variables. This operation is performed by independent functions called *propagators*, which can be seen as operational representations of the constraints provided in the problem specification. A constraint does not necessarily need to correspond to a propagator in a one-to-one relationship.

In each step of the solution search, the constraint programming system applies iteratively the propagators to the sets of possible values until there is a simultaneous fixpoint (that is, no propagator is expected to discard possible values for some variable). In order to illustrate informally the procedure, we can recall the *Send More Money* example introduced in section 3.2. A typical constraint programming system might replace the specified constraints with the following propagators:

- Arithmetic expression → linear equality propagator
- All letters different → all-different propagator
- No leading zeros → two simple inequality propagators

Because of the properties of the propagators, the order in which they are selected for performing propagation does not change the final result of the constraint propagation [30, Ch. 3]. Let us assume that the following arbitrary sequence is executed:

1. The two simple inequality propagators remove the value 0 from the sets of possible values of S and M.
2. The all-different propagator is executed but no value can be removed.
3. The linear equality propagator is executed:
   - M cannot be greater than 1, because there is no combination of values that can cause the sum to be greater or equal to 20000. M must be, therefore, equal to 1.
   - Because $M = 1$, it follows directly that $S = 9$ and $O \in \{0, 1\}$.
4. The all-different propagator is executed again. The corresponding values from the assignments inferred in the previous steps (1 and 9) are removed from the set of possible values of all the other variables. O is thus assigned the value 0.
5. The linear equality propagator is executed again. In order to satisfy the arithmetic expression, E must be in the range between 4 and 7. N, in its turn, must be greater than E.
6. All mentioned propagators are executed again, but no one can discard more possible values. Thus, there is a simultaneous fixpoint and the propagation

is finished. Table 3.1 shows the sets of possible values after each of the given steps.

Step	D	E	M	N	O	R	S	Y
0	{0..9}	{0..9}	{0..9}	{0..9}	{0..9}	{0..9}	{0..9}	{0..9}
1	{0..9}	{0..9}	{1..9}	{0..9}	{0..9}	{0..9}	{1..9}	{0..9}
2	{0..9}	{0..9}	{1..9}	{0..9}	{0..9}	{0..9}	{1..9}	{0..9}
3	{0..9}	{0..9}	1	{0..9}	{0, 1}	{0..9}	9	{0..9}
4	{2..8}	{2..8}	1	{2..8}	0	{2..8}	9	{2..8}
5	{2..8}	{4..7}	1	{5..8}	0	{2..8}	9	{2..8}

Table 3.1: Possible values in each propagation step for *Send More Money*

3.3.2 Search

In most of the problems targeted by constraint programming, propagation is not enough for finding the set of solutions. In those cases, search must be applied by decomposing the problem into simpler subproblems in which propagation is recursively applied. This operation is called *branching*, and is performed by adding new propagators to the subproblems in a way such that the number of possible values in every subproblem decreases. An example of a typical branching strategy could be to select the variable with least possible values (*first-fail*) and define two subproblems with new propagators that split the possible values of the variable.

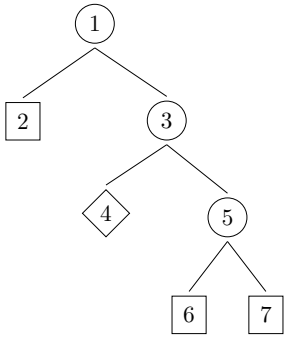
The recursive application of propagation and branching forms a search tree, which can be constructed by following different search strategies, depending on the desired number of solutions and the existence of optimization criteria. Typically, depth-first search is used in order to limit the memory cost [22].

The rest of this section shows search examples for problems with and without optimization criteria. The representation of search trees follows this convention: circles represent nodes that require branching (some variable has more than one possible value), squares represent failed nodes (that is, nodes where some variable has run out of possible values), and diamonds represent solutions. The nodes are labeled by exploration order.

Search for *Send More Money*

In constraint problems without optimization specifications, such as the *Send More Money* problem introduced in subsection 3.2.1, depth-first search is typically used for enumerating all possible solutions. Figure 3.1 shows the search tree and the possible values in each non-failed node when solving the *Send More Money* problem.

The propagators behave like those introduced in subsection 3.3.1 (note that the possible values in the root node coincide with the ones listed in the last row of table 3.1), and the branching strategy is first-fail with smallest value selection.



Node	D	E	M	N	O	R	S	Y
1	{2..8}	{4..7}	1	{5..8}	0	{2..8}	9	{2..8}
3	{2..8}	{5..7}	1	{6..8}	0	{2..8}	9	{2..8}
4	7	5	1	6	0	8	9	2
5	{2..8}	{6, 7}	1	{7, 8}	0	{2..8}	9	{2..8}

Figure 3.1: Search tree and possible values for the *Send More Money* problem

As the figure shows, the problem has a single solution, found in the node 4:

$$\{D \mapsto 7,\ E \mapsto 5,\ M \mapsto 1,\ N \mapsto 6,\ O \mapsto 0,\ R \mapsto 8,\ S \mapsto 9,\ Y \mapsto 2\}$$

Branch-and-bound: search for the constrained *Send Most Money*

As pointed out at the beginning of this section, the same model as for solving the *Send Most Money* problem could be applied to solve constraint problems with optimization, by defining a search strategy that returns all possible solutions to the problem and then selecting the ones that minimize the cost function. In practice, this is usually too costly, and a search strategy called *branch-and-bound* is applied instead.

Branch-and-bound is a strategy for solving optimization problems. Branch-and-bound performs a search in which whenever a solution is obtained, tree branches that can be shown (with the help of a cost function f) to not contain a better solution are discarded (pruned). This strategy fits very naturally in the structure of the defined model: the search is performed as in a modified version of the depth-first strategy. Whenever a solution a is found, a propagator allowing only solutions with better cost than $f(a)$ is added to the set of propagators of the problem. This procedure guarantees that the optimal solution is obtained [30, Ch. 4].

Figure 3.2 shows the search tree and the possible values in each non-failed node when solving a constrained version of *Send Most Money* problem, where the additional basic constraint:

$$E = 7$$

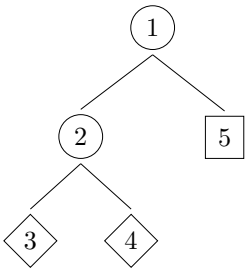
has been added in order to generate a simpler search tree. The same propagators and search strategy as in the previous example has been used. As it can be seen, the first valid combination of values, with cost 10875, is found in the node 3. Immediately after the solution is detected, a propagator representing the following constraint is added:

$$10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y > 10875$$

The next explored node (4) gives a new solution that satisfies this constraint, because its cost is 10876. Therefore, the propagator is updated with the new maximum found cost:

$$10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y > 10876$$

Because the maximum possible value for the cost function under the node 5 is 10876, this node is pruned and the search finishes.



Node	D	E	M	N	O	S	T	Y
1	[2..4]	7	1	8	0	9	[2..4]	[4..6]
2	2	7	1	8	0	9	[3..4]	[5..6]
3	2	7	1	8	0	9	3	5
4	2	7	1	8	0	9	4	6

Figure 3.2: Search tree and possible values for the constrained *Send Most Money* problem

As mentioned above, the solution that maximizes the cost function is given by the node 4:

$$\{D \mapsto 2,\ E \mapsto 7,\ M \mapsto 1,\ N \mapsto 8,\ O \mapsto 0,\ S \mapsto 9,\ T \mapsto 4,\ Y \mapsto 6\}$$

3.4 Available systems

A constraint programming system is a computer system that provides a set of reusable services, such as propagation and search (see section 3.3), for solving constraint problems. The early existence of mature constraint programming systems is one of the most important causes for the width of the application field of this technique [30, Ch. 14]. In this section, a very brief survey of some of the main available constraint programming systems will be given.

The existent systems can be classified depending on if they are standalone (that is, programming languages, typically extensions of some declarative language such as Prolog, or programs by themselves), or they are delivered as libraries written in a mainstream, imperative language such as Java or C++.

3.4.1 Standalone constraint programming systems

The traditional approach to build constraint programming systems has been to extend Prolog systems with constraint capabilities. Two examples of this approach are SICStus Prolog, and ECL^{i}PS^e, which are respectively proprietary and licensed under a variation of the MPL license. It is possible to adapt both systems to perform parallel search, although none of them formally supports it at the time of writing this report.

An alternative approach to the Prolog extension is that of Minion, a system built in C++ that takes a file describing a CSP problem and returns its solution. This lack of interface flexibility allows Minion to be efficient and simple at the same time. Minion is licensed under the GPL license.

3.4.2 Library constraint programming systems

The constraint system that pioneered the idea of a constraint programming library is ILOG Solver [30, Ch. 14], a proprietary system owned by IBM which is accessible from several mainstream imperative languages, such as Java, C++, and .NET.

In the free software category, several libraries such as Choco and JaCoP (implemented in Java) or Gecode (implemented in C++) can be mentioned. While both Choco and JaCoP (licensed respectively under the BSD and AGPL license) set their main focus on ease of use, Gecode (licensed under the MIT license) has openness and efficiency as its main goals, supporting parallel search and showing good performance results in its website.

Chapter 4

The trading system

4.1 Introduction

A financial market is, as mentioned in the introduction of this report, a system that allows buyers and sellers to trade and exchange items of value such as commodities and equity shares. Financial markets have been revolutionized during the last decades by the fast development of computer systems and networks and the introduction of important regulatory changes. Today, almost all financial markets in the world use electronic trading systems, which are subject to challenging requirements in terms of efficiency, flexibility and reliability [34].

TRADExpress, the target system of this master's thesis, is a trading system developed by a Swedish company called Cinnober that aims to meet the demanding requirements of nowadays' financial markets. *TRADExpress* is a complex system developed entirely in Java that supports many different market structures and models, and comprises several components that sum up to millions of lines of code [5].

In this chapter, an overview of the most essential aspects of *TRADExpress* is given, with a focus on those parts that are targeted by the developed random tests. Section 4.2 introduces the essential business modules and architecture of *TRADExpress*. Section 4.3 explains the different types of trade orders that *TRADExpress* allows, and how they are organized in the system. Section 4.4 deals with how trade orders are matched against each other in different trading mechanisms. Finally, section 4.5 gives an overview of the framework that is used in Cinnober for testing *TRADExpress* at the system level.

4.2 Structure

4.2.1 Instruments

A financial instrument (usually called in this context just *instrument*) can be described as an item of value, not necessarily physical, on which trades can be performed. A financial market usually offers different instruments of a specific type to trade with [34]. *TRADExpress* makes it possible to manage several markets, where instruments of different types can be organized in a hierarchical structure with different depth levels [5].

4.2.2 Market participants

Market participants can be defined as the people and institutions that interact in a financial market. Their structure and organization depends on each type of market, but it is generally possible to identify four main types of participants [34]:

- Customers: invest or disinvest their money by buying and/or selling negotiable instruments.
- Brokers: act as agents on behalf of customers, in order to facilitate the access to markets.
- Dealers: trade on own behalf, assuming thus the risk of their operations.
- Market makers: make money by selling to a higher price than they buy to. Add continuity to the market trade flow by ensuring that there is always a counterpart to trade with.

TRADExpress offers an abstract participant model that supports many different configurations. In *TRADExpress*, all participants belong to some member. The type of participant is defined by its user roles, which state the allowed actions and the information access rights. The market administrators are just defined as participants with special roles. By combining different roles, this model makes it possible to define a rich set of participant types [5].

4.2.3 System architecture

TRADExpress has a complex architecture, designed to deal with demanding reliability, flexibility and performance requirements. From the user's perspective, *TRADExpress* can be seen as a system that follows a classical client-server model, where the whole system acts as a service provider to connected clients. However, the system itself is made up of several servers, which can be in its turn replicated.

Furthermore, the architecture of each server is divided into three abstraction layers: the base layer acts as a platform, and includes middleware and persistence services. Above this, the generic trading engine is built with all the standard business components. On top of the system, a customization layer allows to add customer specific business rules and trading strategies [4].

Even though *TRADExpress* comprises many different servers, four main ones provide the basic functionality that this project needs to consider (see figure 4.1):

- Trading Application multipleXer (TAX): handles connections with clients, converts protocols and routes transactions.
- Matching Engine (ME): handles the business functionality of the system, including the processing of trade orders and the information dissemination to different connections.
- Query Server (QS): maintains a copy of the state for different instruments, and serves some client queries in order to lower the load of the ME.
- Common Data (CD): acts as a repository for the system's reference data, containing information, for example, about the defined users and instruments.

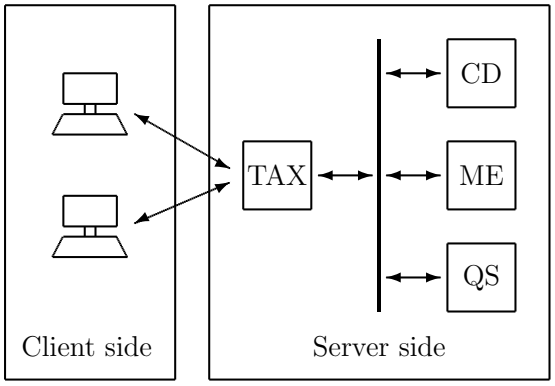


Figure 4.1: Basic architecture of *TRADExpress (adapted from [4])*

TRADExpress can be accessed by clients by using two main protocols: External Messaging Application programming interface (EMAPI) and Financial Information eXchange (FIX). EMAPI is a proprietary protocol which is explicitly designed to access *TRADExpress*, providing low latency and full functionality [4]. FIX is a standard, widely used protocol for trading systems maintained by FIX Protocol Ltd. In both cases, TCP/IP is used at the transport layer when connecting to *TRADExpress*.

4.3 Trade orders

4.3.1 Order books

The main aim of a trading system is to store and match trade orders placed by different participants. This function is performed in a component called *order book*, which is already identified as a core component of every trading system at the introduction of this report. The order book can be seen as a register of the orders that are placed in a certain instrument and have not been matched yet. It is usual to represent it as a table with two columns, respectively containing the buy orders (*bid* side), and the sell orders (*ask* side). In both cases, the orders are sorted by priority, having the top order the highest priority (see figure 4.2).

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

Figure 4.2: Usual structure of an order book representation

Depending on the scope, two main types of order books can be defined [34]:

- Public order books: show the orders from all participants anonymously.
- Private order books: show the orders that a specific participant has placed, and contains information that belongs to the private scope of those orders.

In *TRADExpress*, several order books can be defined for an instrument. When using the EMAPI protocol, information about the state of an order book is obtained by participants by setting subscriptions on the message dissemination flows that are of interest. The following flows are considered in this project [3]:

- Order book event flow: contains events with information about the state of an order book, such as order placements, updates and cancellations.
- Trade flow: contains events with information about the trades which occur (that is, which are *executed*) as a result of order matchings in the order book.

Both flows have public and private versions, containing the private ones extra information about the disseminated events.

4.3.2 Order attributes

The wide range of trading strategies that are applied in financial markets nowadays calls for orders where many different conditions and preferences can be expressed. This is modeled in *TRADExpress* by defining several attributes that can be attached to an order. The main ones are shown in table 4.1.

Attribute	Description
Order book	Specifies the order book where the order will be placed.
Side	Specifies if the order must be placed in the bid or ask side of the order book, i.e., if it is a buy or sell order.
Quantity	Specifies the amount of instrument units that it is desired to trade with. It is possible to specify a minimum quantity, as well as the part of the quantity that is shown in the public order book.
Price condition	Specifies which prices are accepted for trading and how they should be formed: • Market price: the order will be executed at the price set by the market. • Limit price: the order will be executed if the trade price is as good or better than a given one. • Pegged price: the accepted execution price of the order depends on other prices.
Visibility	Specifies if the order is transparent (visible in the public order book), or dark (visible only in the private order book).
Validity period	Specifies the start and end of the period when an order is allowed to match.

Table 4.1: Main order attributes in *TRADExpress (source: [34])*

4.3.3 Order types

The combination of the attributes mentioned in subsection 4.3.2 provides the user of *TRADExpress* with a wide range of available order types, which make it possible to apply different trading strategies. The most common order types are shown in table 4.2. Besides the attributes mentioned in this table, a trade order in *TRADExpress* must always specify the order book, side and quantity q.

Type	Description	Characteristic values
Market	Executed at the price set by the market and then cancelled.	Price condition: market price Validity period: first match only
Limit	Executed if the price is equal or better than p.	Price condition: limit price Limit price: p
Fill-or-kill (FoK)	Either executed at the whole quantity q when inserted or cancelled.	Minimum quantity: q Validity period: first match only
Fill-and-kill (FaK)	Executed when inserted, the remaining quantity (if any) is cancelled.	Validity period: first match only
All-or-none (AoN)	Executed only at the whole quantity q.	Minimum quantity: q
Iceberg	Executed considering the quantity q, but just showing the public quantity q_p.	Public quantity: $q_p, q_p < q$
Pegged	Executed at the best price in its side plus some price offset p_o. Cancelled if there are no orders to peg to.	Price condition: pegged price Price offset: p_o

Table 4.2: Main order types in *TRADExpress (source: [3, 34])*

In order to simplify and avoid ambiguity in the following chapters, the following notation will be applied to the studied order types: a market order with quantity q will be simply represented as q. A limit order with quantity q and limit price p will be represented as q @ p. A minimum quantity limit order with quantity q, minimum quantity q_m and limit price p will be represented as q ($\geq q_m$) @ p. More special attribute values will be explicitly mentioned next to the order. Note that an all-or-none order with quantity q and limit price p can be thus represented as q ($\geq q$) @ p.

It is worth noting that the mentioned types of orders are just some of the most used attribute combinations. The number of possible combinations in *TRADExpress* is practically infinite, and makes the order book a complex system from the design and verification points of view.

4.3.4 Order priority

A common requirement for a trading system is to encourage "generous" prices, high quantities and transparency in the trade orders. In order to guarantee these

principles, *TRADExpress* prioritizes the orders in an order book side by applying the following criteria, sorted by precedence [3]:

1. Price (limit price for limit orders): more generous orders get higher priority
2. Visibility: transparent orders get higher priority
3. Minimum quantity: orders without minimum quantity get higher priority
4. Time: orders inserted before get higher priority

Figure 4.3 shows an example of how the order priority algorithm is applied in *TRADExpress*. In the example, the bid orders b_4, b_3, b_2, b_0 and b_1 are inserted consecutively, and the ask side is empty so they are stored in the order book. Even though it is the first inserted order, b_4 obtains the lowest priority because its price limit is the less generous one. b_2 obtains higher priority than b_3 because it is transparent. b_0 and b_1 obtain higher priority than b_2 because they do not have minimum quantity constraints. Last, b_0 obtains higher priority than b_1 because it is inserted before.

→ **Bid orders**	**Ask orders**
b_0 : 100 @ 10.1 → b_1 : 100 @ 10.1 b_2 : 100 ($\geq$ 100) @ 10.1 b_3 : 100 @ 10.1 (dark) b_4 : 100 @ 10	

Figure 4.3: Order book after consecutively inserting b_4, b_3, b_2, b_0 and b_1

4.4 Order matching

An order book in *TRADExpress* offers several mechanisms to organize the way in which orders match each other to form trades. Matching mechanisms are applied to individual order books, and can be changed during the trading day. The continuous limit order mechanism, called *auto-matching* in *TRADExpress*, is probably the most predominant mechanism used nowadays in financial markets [14, Ch. 2], and for this reason it often concentrates the test efforts and it is as well the target of this project.

From a computational point of view, it is well known that the order matching problem where orders can have minimum quantity constraints is NP-complete [20]. This complexity class includes all matching mechanisms in *TRADExpress* where minimum quantity orders are accepted, auto-matching being among them.

4.4.1 General rules in auto-matching

In auto-matching, trade orders can be inserted and executed on a continuous basis. If the constraints are satisfied, an incoming order is directly matched against the orders that are already inserted in the order book, called *resting* orders. Otherwise, if the incoming order is still valid, it is stored in the order book, becoming a resting order [34].

Match eligibility

The most basic rule which applies to every order matching mechanism is that orders in one side can only be matched against orders placed in the opposite side. While market orders (see section 4.3) do not impose any price constraint for the counter-orders, orders that specify a limit price can only match when the bid limit price is equal or greater than the ask limit price, that is, the bid participant is willing to pay at least the minimum price at which the ask participant is willing to sell [3].

Besides the side and the price limit rule, other constraints can be optionally specified by a trade order. A limit price order with a minimum quantity constraint, for example, cannot match with other orders unless these satisfy the limit price and add quantity enough to satisfy the minimum quantity constraint.

Application of the order priority

In addition to the match constraints that are directly specified in the trade orders, *TRADExpress* defines a priority between orders (see subsection 4.3.4). This priority is applied in auto-matching: an incoming order is allowed to match only if it has higher priority than the other orders resting in its side. More generally, if an order is matched then all orders with higher priority must also have been matched.

A notable exception to the priority rule occurs when orders with minimum quantity constraints are placed in the order book, because orders with less priority can *bypass* them, that is, can be matched first, if the minimum quantity constraint cannot be satisfied [3].

Order validity

Some order types are only valid under certain circumstances. For example, pegged orders are only valid as long as there are orders to peg to in their side, and market orders are only valid during the first match attempt. *TRADExpress* cancels automatically an order as soon as it becomes invalid [3].

4.4.2 General auto-matching procedure

As mentioned above, in auto-matching trade orders are allowed to be executed immediately after being inserted. Every time a trade order is inserted or updated, a two-step matching procedure is triggered: first, a match attempt is performed, where the incoming or updated order is compared with the resting orders, and possibly matched. Then, if *TRADExpress* detects that there are potential matches in the order book, a second match attempt called *re-match* is performed, where all orders from both sides participate. Figure 4.4 illustrates the described execution flow.

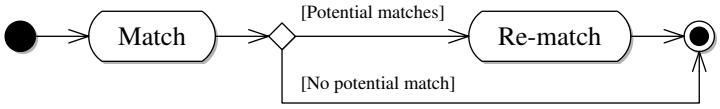


Figure 4.4: UML activity diagram of an order insertion/update in *TRADExpress*

An order cancellation might cause a reconfiguration of the order book such that matches become possible. Because of this, *TRADExpress* always executes a re-match after an order cancellation. Furthermore, changes caused by a single user operation might trigger several order events due, for example, to the recalculation of the limit price of pegged orders. These events are enqueued and generally processed as order updates, following the flow described in figure 4.4.

The following subsections (4.4.3 and 4.4.4) give further detail about each matching step. Appendix B contains an extensive collection of examples of how individual match and re-match situations are solved, which work as a complement for the explanations given here. In the subsequent discussions, as well as in the mentioned examples, a trade between orders b_i and a_j of quantity q at price p will be represented with the following notation:

$$b_i \leftrightarrow a_j : q \ @ \ p$$

4.4.3 Match

In a match, the incoming order is the only one that is considered from its side. The system applies a greedy algorithm, iterating through the resting side in descending order, in order to calculate which resting orders, if any, match the incoming order. The algorithm has a linear complexity, but it does not guarantee to maximize the total traded quantity in order books where orders with minimum quantity constraints are placed.

The basic rule used to determine the price at which a trade is executed is that the resting order sets the price. For example, if an incoming order $b_0 : 10 \ @ \ 25$

matches a resting order $a_0 : 10 @ 22$, the execution price of the trade $b_0 \leftrightarrow a_0$ is set to 22. The execution price of a trade is not allowed to fall out of the visible *Market Best Bid and Offer* (MBBO) of the order book, which is defined as the highest limit price of a resting transparent bid order and the lowest limit price of a resting transparent ask order, excluding orders with minimum quantity constraints. Thus, the execution price of a trade whose resting side would set a price out of the visible MBBO is corrected to the most generous price allowed within it [3].

4.4.4 Re-match

As figure 4.4 shows, a re-match is performed when a match leaves trade orders which can still potentially match, and after order cancellations. The re-match is a more complex algorithm than the match, because it considers all orders from both sides (there is no definition of incoming order) and, unlike the match, it guarantees that the maximum total quantity will be traded, given some constraints.

The main constraint that the system applies in a re-match is that all orders that are matched must have a better or equal price than a reference price called *equilibrium price*, which is is set to the level that maximizes the total traded quantity. The orders that satisfy this constraint are classified into three groups [3]:

- *Must trade* orders: non-minimum quantity orders with limit price better than the equilibrium price.
- *Can trade* orders: non-minimum quantity orders with limit price equal to the equilibrium price.
- Minimum quantity orders.

As their name implies, all *must trade* orders must be completely filled. If two different combinations of orders lead to the same total traded quantity, the one with less absolute imbalance (defined as the absolute value of the difference between the total *can trade* quantity in the bid and ask side) is preferred. Unlike for the match algorithm, the execution price of a trade is always set by the resting side, without requiring it to be inside the visible MBBO.

4.5 The testing framework

4.5.1 The general testing framework

In order to be able to develop automatic tests for *TRADExpress* at a system level, a test automation framework is used at Cinnober. The *TRADExpress* testing framework is developed on top of JUnit, a standard Java testing framework, and provides

support for easily creating participant sessions for both EMAPI and FIX protocols (see subsection 4.2.3), as well as tools for performing standard actions (such as inserting usual trade order types) and verifying the system's response. Figure 4.5 illustrates the layered architecture of the *TRADExpress* testing framework.

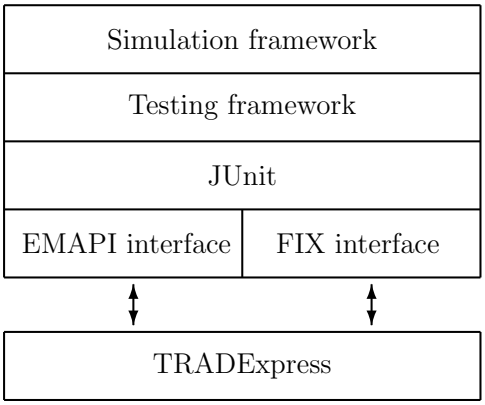


Figure 4.5: Architecture of the *TRADExpress* testing framework

4.5.2 The random testing framework

The facilities offered by the general testing framework are oriented towards the creation of deterministic, systematic test cases. As shown in figure 4.5, a simulation framework is built on top of it. This framework provides a basic structure and random generation tools to construct simulations and random tests. The basic components of a simulation in this framework are the following ones (see figure 4.6):

- Actors: an actor is defined as a collection of actions, where an action comprises always some interaction with *TRADExpress*, like inserting a trade order. Different probabilities can be defined for each action and actor, so that certain actions are more likely to happen than others. An actor usually corresponds (but does not have to) to a *TRADExpress* test participant.
- Oracles: an oracle is defined, in the context of the simulation framework, as a component that is able to anticipate expected results for the system given a certain input. An oracle can receive the system output either directly from the system or be triggered by the actors.
- Simulation core: it is in charge of selecting the actor that must perform an action in each simulation step and keeping track of the duration of the test, which can be defined either in simulation steps or in time.

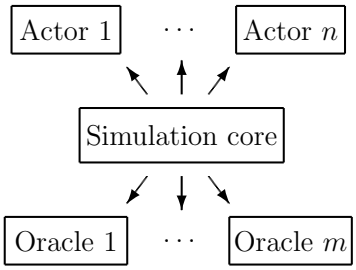


Figure 4.6: Basic components of a simulation

A simulation step in this simulation framework comprises the following sequence:

1. The simulation core chooses the actor that will perform the step action.
2. The chosen actor picks and performs one of its available actions, considering the defined probabilities.
3. Each oracle, triggered either by the chosen actor or by the system's response, calculates the expected system result.
4. The actual system response is compared to the expectations generated by the oracles, and if a failure is detected, it is immediately reported to the simulation core, which stops the simulation and reports the failure to the tester.

Chapter 5

Development of the test system

5.1 Introduction

Chapter 1 exposes the need in the Cinnober's testing process of a system that automatically calculates the expected order book output through trading simulations. The solution to this problem (to develop a random test system with an order book test oracle) is also outlined there, together with the main involved techniques. These techniques, as well as the trading system to test, have been examined in chapters 2, 3 and 4, forming a background framework for discussing the development of the purposed solution with further level of detail.

The fact that this project has been carried out by only one person has made it possible to choose the development methodology with a relatively high degree of freedom. In the first stage, the requirements have been captured and documented. Then, an initial design specification according to the stated requirements has been produced. Finally, the test system has been implemented iteratively, where each iteration comprises adding a new feature, testing the different components individually, integrating them into the test system, executing the test system against *TRADExpress*, and reviewing the design specification.

The following sections of this chapter describe each of the development activities of the prototype random test system. Section 5.2 gives an overview of the components that the purposed solution comprises, and shows their main interactions. Section 5.3 lists and motivates the functional and quality requirements stated for the system. Section 5.4 gives more detail about the final system architecture and the expected behavior of each component. Section 5.5 shows the formal model used by the order book oracle for the computation of the expected trades. Section 5.6 discusses the main implementation decisions and how they relate to the requirements. Last, section 5.7 describes practical aspects of the development process and activities that are not covered in the previous sections.

5.2 System overview

The structure of the developed random test system is mainly based in an integration of the random testing model shown in figure 2.1 into the existing random testing framework outlined in figure 4.6. This scheme comprises the two main components of the test system (the order book actor and the order book oracle), plus the system under test (*TRADExpress*), as shown in figure 5.1. These two main components are driven, in their turn, by the simulation core, which decides which actor is to perform the action in each simulation step, according to the sequence given at the end of section 4.5. The main roles of each component are:

- Order book actor: Picks randomly pre-defined actions to perform on a specific order book of *TRADExpress* and verifies that the trades calculated by *TRADExpress* are the same as the order book oracle expects.
- Order book oracle: keeps the state of a specific order book with the input provided by each actor associated with that order book and calculates which trades correspond to a specific state on request of an actor.

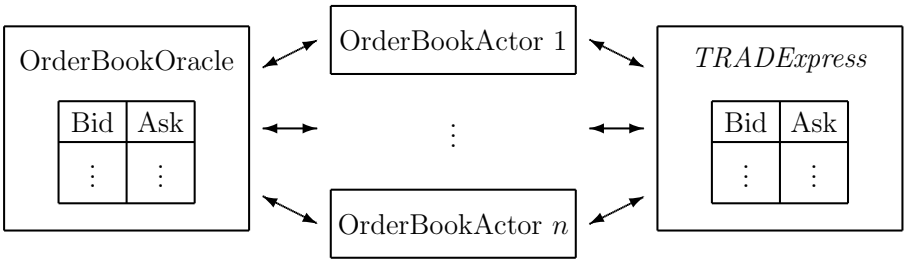


Figure 5.1: Main components and interactions of the system

Although figure 5.1 shows, for the sake of simplicity, a system where a single order book is tested with the help of an oracle, the developed system supports testing an arbitrary number of order books in the same simulation, by simply partitioning the set of actors and defining a new oracle that generates expected results for each order book under test. This allows to reproduce a situation that is closer to the real use of the system and therefore more interesting for testing purposes.

5.3 Requirements

As section 1.4 states, the main aim of this project is to model the business logic that controls the trade order matching in *TRADExpress* to be able to verify its output

through randomly generated trading sequences. This high-level goal is decomposed in this section into several requirements, stated in a solution-independent manner, and prioritized in a must-should-could scale. Subsection 5.3.1 lists the main functional requirements, and subsection 5.3.2 lists the main quality or non-functional requirements.

5.3.1 Functionality

1. **Verification of the *TRADExpress* output**

   (a) The test system must be able to calculate the expected trade events and verify that they are received in the *TRADExpress*' public trade flow (see subsection 4.3.1) after each random action performed on it. For each trade event, at least the following fields will be calculated and verified:

   - Traded quantity
   - Price at which the trade is executed

   (b) The test system could be able to calculate the expected order events and verify that they are received in the *TRADExpress*' public and/or private order book event flow after each random action performed on it.

2. **Action types**

   The test system must be able to perform the following actions on a specific *TRADExpress* order book, by sending the corresponding request message:

   - Insert a new trade order
   - Update an existing trade order
   - Cancel an existing trade order

   The test system should ensure that every generated insert, update and cancel order request is valid according to the *TRADExpress* specification.

3. **Order attributes**

   The test system must be able to generate random valid values from a uniform distribution for, at least, the following order attributes (see subsection 4.3.2) of an insert or update order request:

   - Side
   - Quantity

- Minimum quantity (in fill-or-kill and all-or-none orders)
- Limit price (if the order has a limit price condition)
- Price offset (if the order has a pegged price condition)

The test system should allow the user to specify the range in which random prices and quantities are generated.

4. Order types

The test system must be able to insert and update at least 80 % of the order types listed in 4.2.

5.3.2 Quality

1. Accuracy

(a) The test system must be designed without reference to the code or internal structure of *TRADExpress*.

(b) The test system should use a declarative programming paradigm for the computation of the expected trades.

(c) The test system should use a different programming platform than the one used by *TRADExpress* for the computation of the expected trades.

2. Usability

(a) The test system must display information enough about each simulation step so that a simulation sequence can be unambiguously reconstructed by the user.

(b) The test system must adhere to the style of the automated tests that already exist at Cinnober.

(c) The test system must provide a mechanism to repeat deterministically a specific simulation sequence.

(d) The test system should follow the notation introduced in chapter 4 when displaying information to the user.

3. Maintainability

(a) The test system must be developed in less than 5000 lines of source code (excluding comments and blank lines).

(b) The test system's source code should include a comment for each subroutine that defines its input, purpose and result of calling it.

(c) The test system should include a suite of automated test cases covering its own calculation of expected trades for the order attributes and types specified in 5.3.1.

5.4 Design specification

Two main approaches have been used in the design process: the test system has been designed in an object-oriented fashion both at its highest abstraction level and at a component level for those components that interact with the existing random testing framework. The part of the system that deals with the order matching has been approached in a declarative way, by using constraint programming (see chapter 3), and its design specification, detailed in section 5.5, consists therefore of a constraint problem.

The object-oriented design methodology has been chosen due to two main reasons:

- The requirement of adhering to the automated test development style used in Cinnober (see subsection 5.3.2), and the ease of interfacing the existing random testing framework.
- The well-known benefits that object-oriented offers, particularly modularity and simplicity to map the concepts introduced in chapter 2 such as *test oracle*.

In its turn, constraint programming has been chosen for the design and implementation of the trade calculation logic because, as mentioned in the introduction, it allows to model this complex problem in a formal mathematical notation that helps to close the gap between requirements and implementation, and removes a big part of the implementation effort, which is shifted to the constraint programming system. The constraint programming models are encapsulated in object-oriented classes, and are specified in section 5.5.

The rest of the section is organized as follows: subsection 5.4.1 shows the design of the test system by describing the main classes and their interactions and responsibilities. Then, the design is specified for the three main aspects of random testing identified in chapter 2: the input (subsection 5.4.2), the test oracle (subsection 5.4.3) and the output (subsection 5.4.4).

5.4.1 Overview of the test system

Section 5.2 identifies the actor and the order book oracle as the main components of the test system. The behavior of both of them is driven by the simulation core, which is in charge, as mentioned in the overview, of selecting the actor to perform an action in each simulation step. The order book actor gets the expected trade events from the oracle, and compares them to what it actually gets from *TRADExpress*. Figure 5.2 shows a sequence diagram specified in Unified Modeling Language (UML) with the sequence that each simulation step follows.

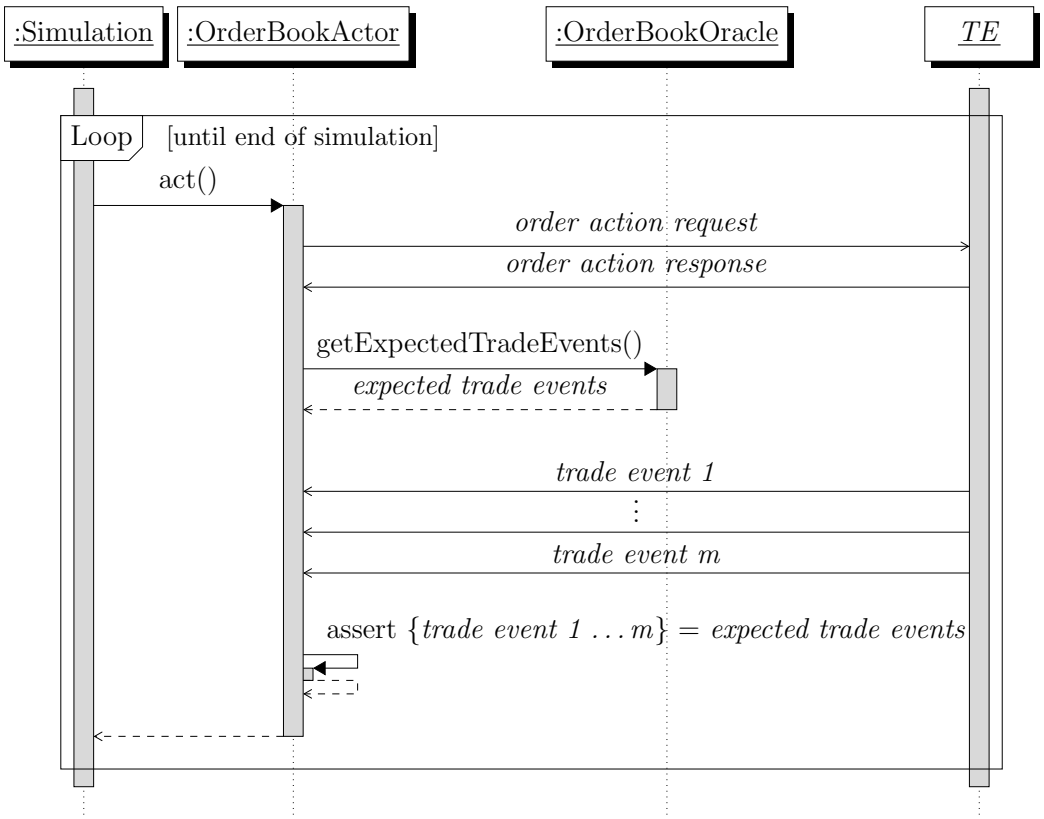


Figure 5.2: UML sequence diagram of an action performed on *TRADExpress* (*TE*)

The order book oracle relies on an inner component, the order book model, to hold a representation of the order book under test and to calculate the expected trade events in a given state. In order to perform this task, the order book model uses finally the order matcher and re-matcher spaces, which, together with their base class, contain the constraint programming model that defines the order matching calculations. Figure 5.3 shows a simplified UML class diagram with the main classes of the test system and their relationships.

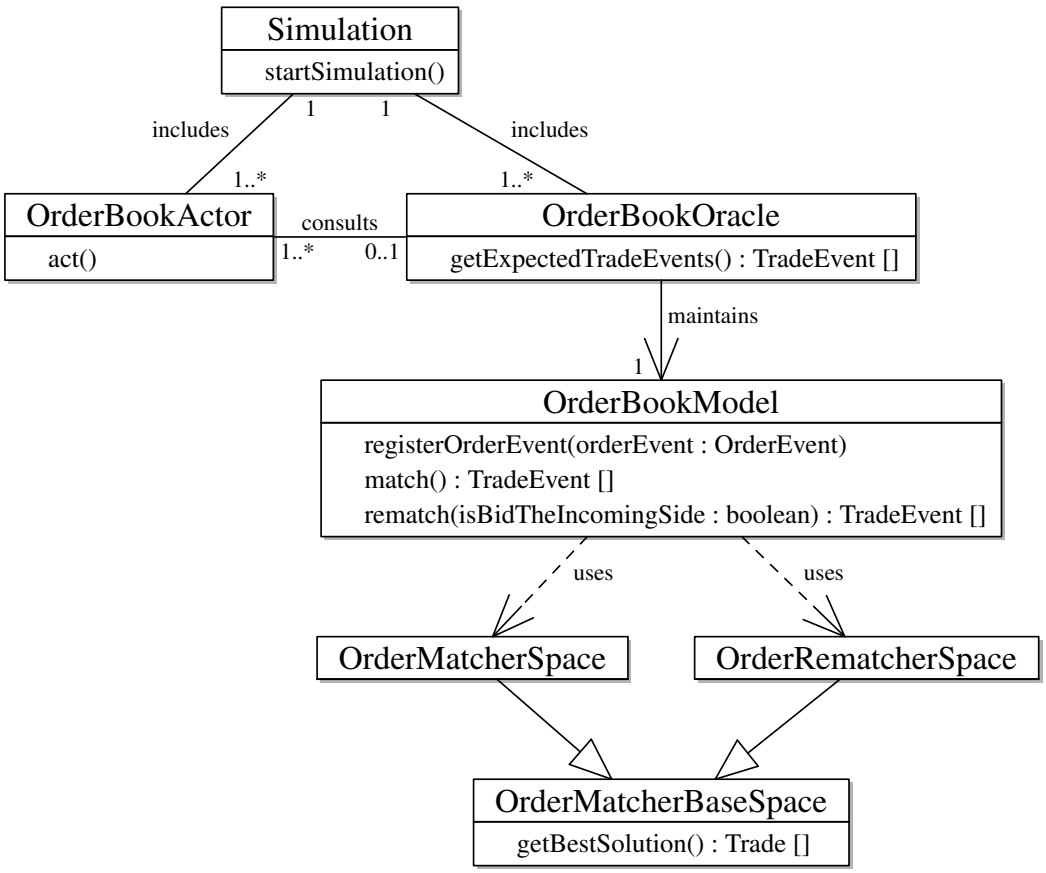


Figure 5.3: Simplified UML class diagram of the test system

5.4.2 The input

As it can be inferred from the functional requirements (see subsection 5.3.1), the input domain for the test system is focused in those actions that modify the state of an order book, either by inserting, updating, or cancelling orders in it. The subdomains for this case can be seen as the different combinations between purposed actions (functional requirement 2) and considered order types (functional requirement 4). As explained in 4.3.3, there are many of these combinations, so an strategy must be adopted in order to select a representative subset of them. Figure 5.4 shows a taxonomy tree of trade orders with the most usual combinations of attributes (see functional requirement 3). The leaves of the tree contain the trade order types that have been chosen as representatives of each class. Note that, for the sake of clarity and simplicity, iceberg orders have been discarded in this first version of the test system.

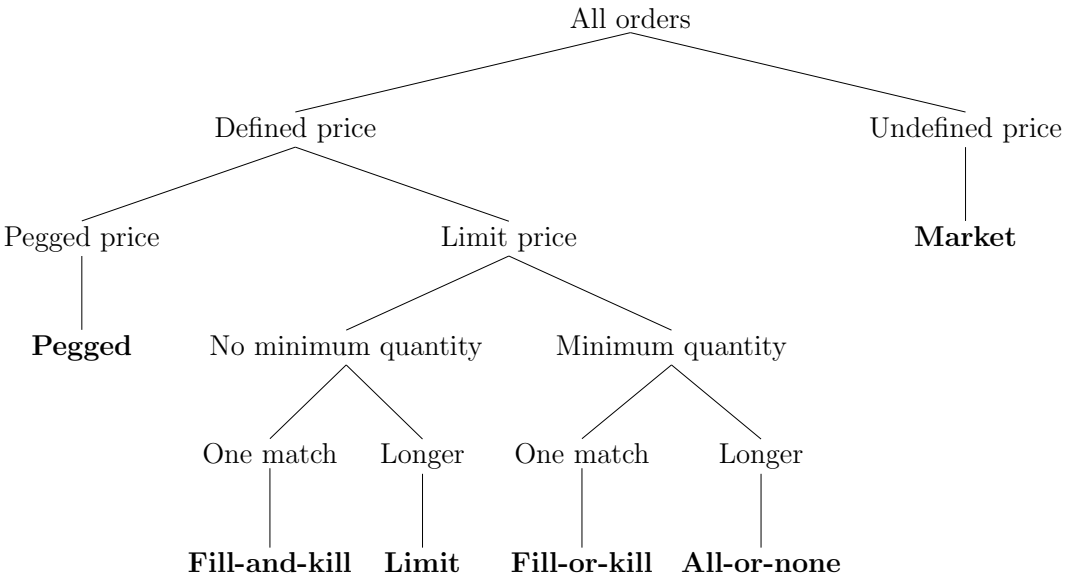


Figure 5.4: Trade orders taxonomy for the input domain modeling

The class in charge of performing the input actions is, as mentioned above, the order book actor. The probability of an actor and action being chosen, as well as the number of actions (which defines the size of the input sequence) and the actors that participate can be specified by the user before running the random test. This allows the user to tune the input distribution depending on the goals of the random test (see subsection 5.4.2). An order book actor is able to perform the following actions:

- **Insert a market order**: insert a market order with random side and quantity.
- **Insert a limit order**: insert a limit order with random side, quantity and limit price.
- **Insert a fill-or-kill order**: insert a fill-or-kill order with random side and quantity, minimum quantity set to the quantity and random limit price.
- **Insert a fill-and-kill order**: insert a fill-and-kill order with random side, quantity and limit price.
- **Insert an all-or-none order**: insert an all-or-none order with random side and quantity, minimum quantity set to the quantity and random limit price.
- **Insert a pegged order**: select a random side. Insert, if there are orders to peg to in the side, a pegged order pegged to the best side price, with random quantity and random price offset within a fixed range.

- **Update an order**: select randomly an own order from the order book. If there is any, update it with a new random quantity, and new random values for each relevant attribute depending on the order type.
- **Cancel an order**: select randomly an own order from the order book. If there is any, cancel it.

All quantities and prices can be limited to a certain range, and the decimal precision of prices can be also specified.

5.4.3 The order book oracle

As chapter 2 explains, there are different types of oracles that vary in completeness, complexity and other characteristics. The developed order book oracle can be classified as a true oracle, as it aims to reproduce the exact expected trade flow output for the specific input sub-domain explained in the previous subsection.

The main challenge in the design of the order book oracle is to implement the functionality stated in the functional requirement 1 without sacrificing simplicity and independence (see quality requirements 1 and 3). In order to reduce the complexity, an inner class called *order book model* has been defined. This class represents the state of the order book, and contains methods to calculate expected trades out of that state (see section 5.5), and a single method to change its state. It is the responsibility of the order book oracle to keep the order book model updated with the corresponding order events that are received or inferred from the trade events.

The most complex operation that the order book oracle performs is the calculation of the expected trade events that are derived from a user order action, such as the insertion of a new order. Figure 5.5 shows an UML sequence diagram with the interactions that take place between *TRADExpress*, the oracle and the order book model in these calculations. The sequence can be summarized in the following steps:

1. The oracle receives all the order events from *TRADExpress* and initializes a queue of order events with the one corresponding to the user action.
2. The oracle extracts the oldest order event from the queue, and updates the order book model with it.
3. The oracle gets the expected trades from performing a match in the order model, and updates it with the order events that are derived from those trades and the order events that are expected after a match.
4. The oracle gets the expected trades from performing a re-match in the order model, and updates it with the order events that are derived from those trades and the order events that are expected after a re-match.

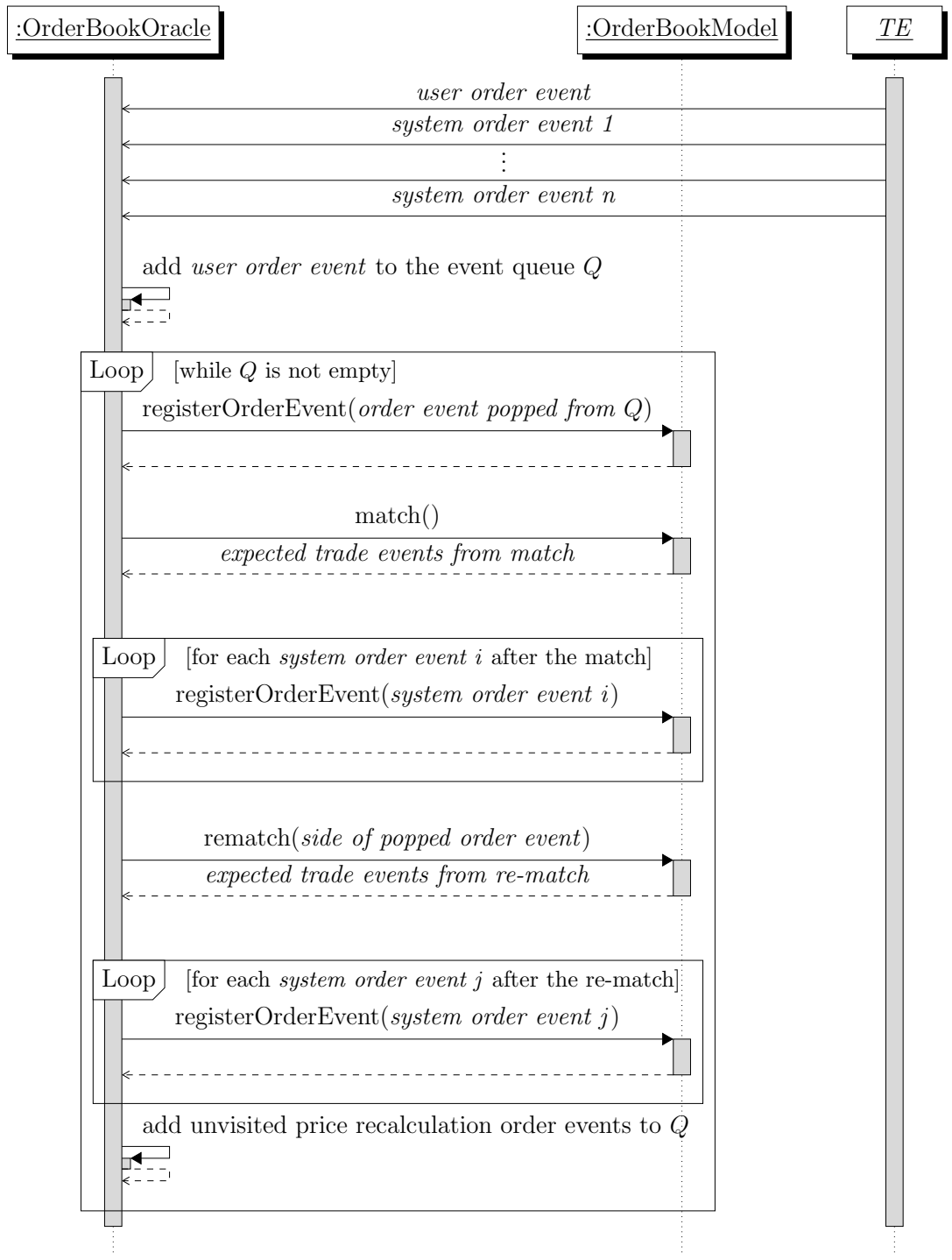


Figure 5.5: UML sequence diagram of the calculation of the expected trade events

5. The oracle adds all possible price recalculation events that are derived from the state of the order book model and have not been added before to the queue, and if the queue is not empty goes to step 2 again.

6. The oracle returns the set of all calculated expected trades.

The algorithm 1 shows the process described above in a more formal way. The key data structures are the set of trade events to return F, the queue of order events that wait to be processed Q and the set V of order events that are waiting in Q or have already been processed. After the order book model has calculated which trade events should be expected from a match or a re-match, derived order events must be calculated in order to update the state of the order book model:

Input: A set of order events E corresponding to the latest user action.
Output: A set of trade events T (expected result to the latest user action).

$T \leftarrow \emptyset$
$e_u \leftarrow$ *user order event* $\in E$
$Q \leftarrow \{e_u\}$ `// Q is a queue of order events to process`
$V \leftarrow \{e_u\}$ `// V is a set of visited order events`
while $Q \neq \emptyset$ **do**
  $e_o \leftarrow$ *oldest event* $\in Q$
  $Q = Q - \{e_o\}$
  register $\{e_o\}$ in the order book model OB
  $T \leftarrow T \cup \{$*expected trades from a match in* $OB\}$
  register $\{e \in E : e$ *derived from the match expected trades*$\} \cup$
    $\{e \in E : e$ *is a one match order cancel event*$\} \cup$
    $\{e \in E : e$ *is a pegged order cancel event*$\}$ in OB
  $T \leftarrow T \cup \{$*expected trades from a re-match in* $OB\}$
  register $\{e \in E : e$ *derived from the re-match expected trades*$\} \cup$
    $\{e \in E : e$ *is a pegged order cancel event*$\}$ in OB
  $E_P \leftarrow \{e \in E : e$ *is a pegged order recalculation event*$\}$
  $Q \leftarrow Q \cup (E_P - V)$
  $V \leftarrow V \cup E_P$
end
return T

Algorithm 1: Calculation of the expected trade events by the test oracle

- Order events derived from expected trades: order events that represent the cancellation of completely filled orders and the update of the quantity of partially filled orders.

- One match order cancel events: order events that represent the cancellation of orders with the validity period set until the next match. These cancellation occurs always after the first match attempt.
- Pegged order cancel events: order events that represent the cancellation of pegged orders that do not longer have any order to peg to in their side.
- Pegged order recalculation events: order events that represent the update of pegged orders because of a best "peggable" price change in their side.

As it can be seen in the pseudo-code given in algorithm 1, the operation of calculating expected trades and updating the state of the order book iterates while the queue Q is not empty. It is important to ensure that this condition always evaluates true after a finite number of iterations and the while-loop terminates. A completely rigorous termination proof would require many formalization steps that fall out of the scope of this project, so the termination is proven in a more informal style.

Informal proof of while loop termination. Let us consider the set of trade events T_i, the queue of order events Q_i, the set of pegged order recalculation events E_{P_i} and the sum of the quantity of all orders in the order book $|q_i|$ at the end of the iteration $i,\ i > 0$.

In the iteration i, one of the following disjoint cases takes place:

- $|T_i| = |T_{i-1}|$. Then no order is filled and $|q_i| = |q_{i-1}|$. Because of this, the best "peggable" prices in the order book are not updated and $E_{P_i} = \emptyset$, and hence no new order event is added to Q_i, and $|Q_i| < |Q_{i-1}|$.
- $|T_i| > |T_{i-1}|$. Then some order has been filled and $|q_i| < |q_{i-1}|$, which might lead to new price recalculation events resulting in $|Q_i| \geq |Q_{i-1}|$.

Therefore the sequence of pairs of $|q_i|$ and $|Q_i|$ in successive iterations follows a lexicographical order:

$$\langle |q_i|, |Q_i| \rangle \underset{lex}{<} \langle |q_{i-1}|, |Q_{i-1}| \rangle, \quad i > 0$$

As the relation $<$ is well-founded for non-negative integers (which is the domain of both elements in the tuple), the lexicographical order is well-founded and the given sequence is finite. Hence, the while-loop terminates.

□

5.4.4 The output

Section 2.4 deals with two important aspects related to the output of a random test: test case reduction, which refers to the task of isolating a failure found in a simulation sequence, and reliability estimation, which offers techniques to infer reliability models from statistical analysis of the random test output.

Test case reduction can be a very tedious task, so it is desirable to automate it, by using techniques as the one suggested in the mentioned section. However, these techniques require an adaptation of the test system to make it able to produce machine-readable logs and replay subsets of a given simulation sequence, which falls out of the scope of this project. In order to isolate a failure, therefore, manual analysis of the failing simulation sequence must be performed, with help of the test output and the possibility to deterministically repeat the whole simulation sequence (see quality requirement 2).

Reliability estimation requires, as mentioned in the end of section 2.4, the use of a realistic operational profile in the test input distribution. The test system input is highly configurable and accepts different input distributions, but the creation of a realistic operational profile is a demanding task, out of the scope of this project, so it will not be considered as a part of the development test system.

A third output aspect considered in the design is the collection of statistics in test runs. The following statistics are collected and shown to the user at the end of a test case:

- Total match hits (that is, match attempts that return some expected trade) and total match expected trades.
- Total re-match hits (that is, re-match attempts that return some expected trade) and total re-match expected trades.

These statistics can be useful to get an idea of which components of the system under test have been exercised most in each simulation.

5.5 Computation of the expected trades

As mentioned in section 5.4, the problem of computing the expected trades has been approached in a different way to the rest of the test system. The order book model, whose state is driven by the test oracle (as explained in subsection 5.4.3), calculates the expected trades of a match or re-match operation by defining and solving a constraint satisfaction problem with its state as input data.

As section 4.4 explains, *TRADExpress* combines the use of two different strategies when running the auto-matching mechanism: match and re-match. As both

strategies are similar in an important number of points, the common elements have been pulled up into a common constraint satisfaction problem, represented in the figure 5.3 by the class *OrderMatcherBaseSpace* and defined in subsection 5.5.1. Each derived problem (represented by the classes *OrderMatcherSpace* and *OrderRematcherSpace* and defined respectively in subsections 5.5.2 and 5.5.3) inherits thus the variable definitions and constraints specified in this class, adding only the new variables and constraints that are specific to the problem.

In order to simplify the constraint problem as much as possible, each order that is used as input is modeled by only three attributes: quantity q, minimum quantity mq and limit price lp. These attributes get always a value, even if no direct mapping is possible from the original *TRADExpress* order. The quantity is directly filled with the value from the original order, and the minimum quantity and limit price of each order model are assigned according to table 5.1.

Type	***mq***	***lp***
Market	0	best price in order book + 1
Limit	0	lp
Fill-or-kill	q	lp
Fill-and-kill	0	lp
All-or-none	q	lp
Pegged	0	best price in its side + offset p_o

Table 5.1: Modelling of *TRADExpress* orders for the order matching problem

The calculation of the execution price of every trade varies in each of the derived matching steps. Even though it is performed after the matching result is obtained and is not actually part of the constraint problem, it is calculated by the *OrderMatcherSpace* and *OrderRematcherSpace* classes, and it can be formalized and expressed in terms of the same notation used in the constraint problems. For these reasons, the formal model for the calculation of the execution price is attached as well at the end of subsections 5.5.2 and 5.5.3.

5.5.1 The common order matcher

The common order matcher is not a complete order matcher in itself, as it does not define any cost function and thus multiple solutions are allowed, but it provides a framework in which the two order matchers are based. It includes the integer variables that represent the quantity matched between different orders, as well as some auxiliary variables that are useful in the construction of more advanced constraints.

Furthermore, it includes constraints that set the basic limitations in how quantities can be distributed.

Input data

An order book with $n \geq 1$ bid and $m \geq 1$ ask orders sorted descendingly by priority:

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_i}, mq_{b_i} and lp_{b_i} respectively represent the quantity, minimum quantity and limit price of the bid order b_i. For the sake of simplicity, we define the set of bid orders $B = \{b_i : 0 \leq i < n\}$, and the set of ask orders $A = \{a_j : 0 \leq j < m\}$.

Variables and domain

The following non-negative integer variables represent all possible trades between the given orders:

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

where tq_{b_i,a_j} represents the quantity traded between the bid order b_i and the ask order a_j. The auxiliary variables tq_{b_i}, tq_{a_j} and tq respectively represent the total traded quantity of the bid order b_i, the ask order a_j and between all orders.

Constraints

Auxiliary variables definition

tq_b represents the total traded quantity of the bid order b:

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

tq_a represents the total traded quantity of the bid order a:

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

tq represents the total traded quantity between all orders:

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

Limit price: a bid order b cannot trade with an ask order a with higher limit price:

$$lp_b < lp_a \implies tq_{b,a} = 0 \quad \forall \langle b, a \rangle \in B \times A$$

Individual order quantity: an order x can only trade up to its quantity:

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

Traded quantity: a bid order b and an ask order a can only trade up to the minimum of their quantities:

$$tq_{b,a} \leq \min(q_b, q_a) \quad \forall \langle b, a \rangle \in B \times A$$

Minimum order quantity: an order x must trade either 0 or a quantity greater than its minimum quantity:

$$tq_x \geq mq_x \vee tq_x = 0 \quad \forall x \in B \cup A$$

Order priority: a limit order without minimum quantity constraint cannot be bypassed by other order with less priority:

$$tq_{b_i} < q_{b_i} \wedge \ mq_{b_i} = 0 \implies tq_{b_j} = 0 \quad \forall \langle b_i, b_j \rangle \in B^2 : i < j$$

$$tq_{a_i} < q_{a_i} \wedge \ mq_{a_i} = 0 \implies tq_{a_j} = 0 \quad \forall \langle a_i, a_j \rangle \in A^2 : i < j$$

5.5.2 The order matcher

The order matcher represents the most common matching strategy used in auto-matching. The incoming order is the only order allowed to match in its side, and the matching must be performed sequentially, starting from the order with highest priority at the resting side.

Input data

- The order book inherited from the common model.
- The incoming order c and its side $C : c \in C,\ C \in \{B, A\}$.

Variables and domain

The order matcher uses only the traded quantity variables inherited from the common model.

Cost function

The solution must maximize the total traded quantity:

$$maximize\ tq$$

Constraints

Incoming order: c is the only order allowed to trade in its side:

$$tq_x = 0 \ \ \forall x \in (C - \{c\})$$

Sequential matching: an order in the resting 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 less priority:

$$c \in A \ \wedge \mathrm{atq}(b_i) + q_{b_i} \leq q_c \ \wedge tq_{b_i} < q_{b_i} \implies tq_{b_j} = 0 \ \ \forall \langle b_i, b_j \rangle \in B^2 : i < j$$

$$c \in B \ \wedge \mathrm{atq}(a_i) + q_{a_i} \leq q_c \ \wedge tq_{a_i} < q_{a_i} \implies tq_{a_j} = 0 \ \ \forall \langle a_i, a_j \rangle \in A^2 : i < j$$

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

$$\text{atq}(x_0) = 0$$
$$\underset{i\,>\,0}{\text{atq}(x_i)} = \begin{cases} \text{atq}(x_{i-1}) + q_{x-1}, & \text{atq}(x_{i-1}) + q_{x-1} \leq q_c \\ \text{atq}(x_{i-1}), & \text{otherwise} \end{cases}$$

Trade price

The execution price tp at which two orders trade is set by the resting side, and it must fall into the visible spread, if this can be defined:

$$c \in B \implies tp_{b,a} = \begin{cases} \max_{x \in C'} lp_x, & |C'| > 0 \wedge \max_{x \in C'} lp_x > lp_a \\ lp_a, & \text{otherwise} \end{cases}$$
$$\forall \langle b, a \rangle \in B \times A : tq_{b,a} > 0$$

$$c \in A \implies tp_{b,a} = \begin{cases} \min_{x \in C'} lp_x, & |C'| > 0 \wedge \min_{x \in C'} lp_x < lp_b \\ lp_b, & \text{otherwise} \end{cases}$$
$$\forall \langle b, a \rangle \in B \times A : tq_{b,a} > 0$$

where C' is the set of resting visible orders in the incoming side:

$$C' = \{x \in C : mq_x = 0\} - \{c\}$$

5.5.3 The order re-matcher

The order re-matcher is applied when there are potential trades that the order matcher cannot solve. New variables, including auxiliary ones, are defined, and the cost function becomes more complex, as there is a hierarchy of criteria defined in order to chose between different order combinations. As the order re-matcher is always applied after the order matcher, the incoming order is not considered (it could have been completely filled by the order matcher). However, the incoming side is still taken into account for the calculation of the execution price.

Input data

- The order book inherited from the common model.
- The incoming side $C \in \{B, A\}$.

Variables and domain

- The traded quantity variables inherited from the common model.
- The 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 $\{lp_x : x \in B \cup A\}$.
- The auxiliary integer variable im, which represents the imbalance in a calculated matching.
- The auxiliary integer variables it_{max}, it_{min}, jt_{max} and jt_{min}, which represent the indexes of the minimum intervals that contain orders that trade.
- The auxiliary non-negative integer variables $tq_0, \ldots, tq_{n+m-1}$, where tq_k represents the total traded quantity between orders whose indexes sum k.

Cost function

The solution must lexicographically maximize the following tuple of integer variables (note that the expressions with negative sign are to be minimized):

$$maximize\ \langle tq,\ -|im|,\ -(it_{max} + jt_{max}),\ (it_{min} + jt_{min}),\ -eqp,\ tq_0,\ \ldots,\ tq_{n+m-1}\rangle$$

Constraints

Auxiliary variables definition

im represents the difference between the total *can trade* quantity in the bid and ask sides (see subsection 4.4.4):

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

it_{max}, it_{min}, jt_{max} and jt_{min} respectively represent the maximum and minimum indexes, for the bid and ask side, of an order that trades:

$$it_{max} = \max(\{-1\} \cup \{i : b_i \in B \wedge tq_{b_i} > 0\})$$

$$it_{min} = \min(\{n\} \cup \{i : b_i \in B \wedge tq_{b_i} > 0\})$$

$$jt_{max} = \max(\{-1\} \cup \{j : a_j \in A \wedge tq_{a_j} > 0\})$$

$$jt_{min} = \min(\{m\} \cup \{j : a_j \in A \wedge tq_{a_j} > 0\})$$

tq_k represents the total traded quantity between orders whose indexes sum k:

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

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

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

$$lp_a > eqp \implies tq_a = 0 \ \ \forall a \in A$$

Must trade quantity: the total *must trade* quantity in both sides must be completely filled (see subsection 4.4.4):

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

Trade price

The execution price tp at which two orders trade is set by the resting side:

$$tp_{b,a} = \begin{cases} lp_b, & C = A \\ lp_a, & C = B \end{cases} \qquad \forall \langle b, a \rangle \in B \times A : tq_{b,a} > 0$$

5.6 Implementation

The implementation decisions taken in this project have a great impact in the fulfilment of the requirements stated in section 5.3, specially the quality ones. They can be divided in two parts, analogously to how the specification of the design is organized: those related to the actor, oracle and order book model are detailed in subsection 5.6.1; those related to how the order book model computes the expected trades are detailed in subsection 5.6.2.

5.6.1 Actor, oracle and order book model

Language and platform

With the aim of reusing existing work and reduce integration and learnability costs (see quality requirement 2), the actor, oracle and order book model component have been implemented as regular Java classes, and are integrated into the existing simulation framework (see figure 4.5) at the same level as the existing components. Appendix A includes the Javadoc documentation of the main classes.

In order to minimize the complexity and increase the maintainability and readability of the Java code (see requirement 3), high-level constructions as the following ones have been applied whenever possible:

- The order predicates defined in the orders taxonomy shown in figure 5.4 have been directly implemented as classes that implement the *Predicate* interface of the Apache Commons, a library of reusable Java components. These predicates are used, for example, for filtering orders in collections.
- The logic that controls the order priority is encapsulated as a class that implements the *Comparator* interface of the Java platform. The order book model is sorted according to this comparator. Other comparators are defined in order to avoid code replication and increase the readability of the code.

User interface

The development of an advanced user interface falls out of the scope of this project. The test system simply uses the console standard output to display the action performed, the state of the order book model and the expected and actual trades in each simulation step, as stated in requirement 2.

The random number generation is centralized and initialized by a seed, which can be specified by the user in the test script. As the state of the tested order books in *TRADExpress* is restarted between each run, the user can repeat a certain test case of interest by simply specifying that the same seed should be used in the test script. Figure 5.6 shows an example test script with only one actor and one oracle where the initialization seed is set to 1000 and the simulation length is set to 100 steps.

Furthermore, the test oracle returns the expected trades as matcher objects, which can be directly used by the constructions implemented in the random testing framework for performing comparisons.

```
@Test
public void simpleScenarioWithOneActor() {

  setSeed(1000L);
  setExecutionLength(100L);

  OrderBookOracle tOracle = new OrderBookOracle(this, MOPS_USER, mOrderBook);
  addOracle(tOracle);

  addActor(new OrderBookActor(this, "PARTICIPANT", USER, mOrderBook, 100)
      .useOrderBookOracle(tOracle)
      .setPriceLimits(10, 1000)
      .setQtyLimits(10, 100)
      .weightAction(OrderBookActor.INSERT_LIMIT_ORDER, 30)
      .weightAction(OrderBookActor.INSERT_ALL_OR_NONE_ORDER, 30)
      .weightAction(OrderBookActor.INSERT_FILL_AND_KILL_ORDER, 20)
      .weightAction(OrderBookActor.CANCEL_ORDER, 10)
  );

  startSimulation();

}
```

Figure 5.6: Example of a test script with one oracle and one actor

5.6.2 Computation of the expected trades

As mentioned in the previous sections, the components in charge of computing the expected trades given a model of a certain state of the order book are the *OrderMatcherBaseSpace* and the derived *OrderMatcherSpace* and *OrderRematcherSpace*. These components are defined as classes and encapsulate constraint problems to be solved by a constraint programming system. As section 3.4 shows, there are many constraint programming systems that could potentially solve the order matcher and re-matcher problems specified in section 5.5. Gecode has been chosen out of them for the following reasons, among others:

- It is delivered as a C++ library, which makes the integration with the test system easy.
- Its openness reduces the risk that the lack of a needed feature will block the development.
- I have previous experience with it, which has reduced the development time.

The Gecode models, written in standard C++, have been implemented as a shared library, and are connected to the rest of the test system by using the Java Native Interface (JNI). In this way, the quality requirement 1 has been satisfied. As for the Java code, the C++ code makes an intensive use of high-level constructions, for example by applying methods and structures from the C++ Standard Library

and using the direct modeling support ([32, Ch. 6]) provided by Gecode whenever possible.

5.7 Practical aspects of the development process

The previous sections have introduced the design specification and the main implementation decisions taken in the development of the test system. This section sets one more level of abstraction and deals with the development process itself, the methods applied, the organization of the available time and the main faced issues.

5.7.1 Organization of the development process

As the initial time plan states, the development phase of this master's thesis has been carried out in a period of 8 weeks, between October and November 2009. As stated in the introduction to this chapter (section 5.1), the test system has been implemented iteratively, adding support for testing, generation, modeling and verification of an order type in each iteration. The following practices have been applied to the development process:

- Explicit statement of the requirements and prioritization by means of a backlog.
- Test-driven development applied to the trade calculation subsystem, as stated in the quality requirement 3. Appendix B includes a selection of the most illustrative developed test cases.
- Version control of the code base, documentation and programming diary.

Figure 5.7 shows a timeline with the main events of the development period. The first working version, available since week 42 and enlarged during weeks 43 and 44, uses only the order matcher specified in subsection 5.5.2. Week 45 was spent in trying to generalize the order matcher model so that it could support all situations with all-or-none (AoN) orders involved. By the end of week 45, splitting up the initial order matcher into two matchers (the matcher and the re-matcher) was identified as the only possible option to reproduce the expected behavior of *TRADExpress*. Week 46 and 47 were taken to develop the order re-matcher, and the final version supporting all-or-none orders was available on week 48. As a consequence of the design changes, the development of support for all-or-none orders has taken practically the same time as the development of support for all the other order types together.

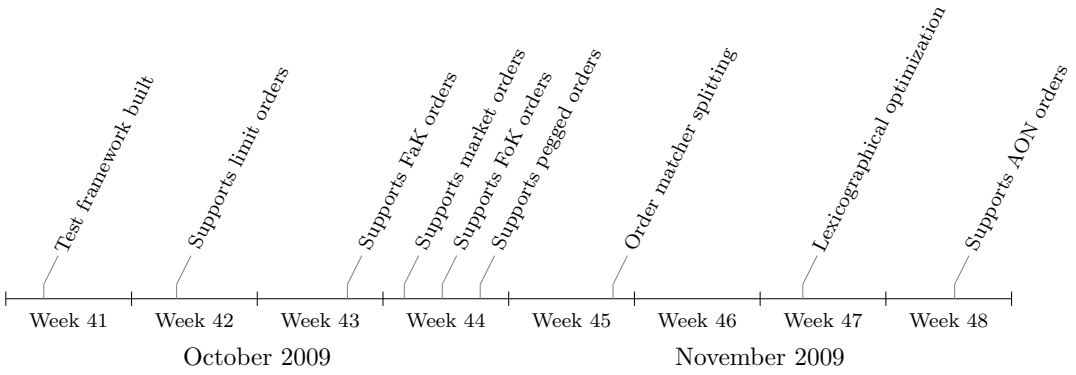


Figure 5.7: Timeline of the development process

5.7.2 Issues

Trade priority in the order re-matcher

Figure 5.7 above shows the insight of using lexicographical optimization for the order re-matcher as a main event in the development process. This is translated into the following cost function in the final re-matcher problem (from subsection 5.5.3):

$$maximize\ \langle tq,\ -|im|,\ -(it_{max} + jt_{max}),\ (it_{min} + jt_{min}),\ -eqp,\ tq_0,\ \ldots,\ tq_{n+m-1}\rangle$$

While the first two elements of the tuple tq and $-|im|$ are easily inferred from the requirements database, the rest of elements are added only on the basis of observations of actual results from *TRADExpress*, because it has not been possible to infer the information found in the requirements database and example scenarios with enough precision to be able to formalize it. Therefore, the order re-matcher might produce false positive results (that is, expected trades that do not match the actual results) in those order book states that are complex enough to contain several possible order combinations with the same total trade quantity tq and absolute imbalance $|im|$. Some of such cases have been detected, documented and added to the test suite.

As an order re-matcher attempt can only succeed when all-or-none orders are inserted into an order book, and furthermore the order book must reach a quite complex situation for this issue to be reproduced, its impact to the test process can be seen as limited. In order to evaluate the frequency of failure of the test oracle due to this issue, a simple experiment has been performed: random test cases with an input sub-domain designed to maximize the number of complex order re-matcher situations (exclusive insertion of all-or-none orders with a constant limit price) have

been run. In these cases, no more than 10% of successful re-matches (that is, those with $tq > 0$) have been revealed as false positives.

Domain of the order attributes

In order to avoid precision problems associated with the use of floating point numbers, *TRADExpress* uses Java long integers for representing prices and quantities, even if the formers are presented to users as real numbers [4]. Standard C++ and Gecode, in their turn, only support standard integers, which typically use half the bytes of Java long integers. This type mismatch limits the input domain: quantities and prices must be limited so that they fit into standard C++ integer variables.

On the other hand, attempts to calculate the expected trades in certain order book configurations (with many orders and high quantities) might end up in a search explosion when solving the associated constraint problem. This issue limits the input domain as well, specially when including all-or-none orders, which generally add more combinatorial richness to the constraint problem.

Even though these issues prevent the exploration of the whole domain of the order attributes, situations of equivalent complexity from the order matching perspective can be generated with limited price and quantity values, and thus the same coverage can be obtained as if we were able to explore the whole domain of these attributes. The impact of both issues to the potential coverage of the test system is considered to be thus limited again.

Chapter 6

Results and analysis

6.1 Introduction

This chapter analyzes, with the help of the collected results, in which degree the chosen solution fulfills the problem stated in section 1.3. The discussion is organized by showing results and assessing the achievement of each of the goals presented in subsection 1.4.1:

1. Evaluate the suitability of constraint programming as a modeling technique for solving the order matching problem.
2. Evaluate how the introduction of a true test oracle contributes to the improvement of the trading system's testing process.

In order to assess the degree of achievement of the first goal, the main characteristics of the order book oracle, such as its completeness and maintainability will be examined by considering, whenever possible, measures like the ones anticipated in subsection 1.4.2. The success of the second goal will be analyzed, in its turn, by looking at the defects found in *TRADExpress* during the development and execution of the implemented test system.

The chapter is organized in two main parts, each of them respectively related to the goals listed above: the first part (section 6.2) analyzes the main characteristics of the developed test oracle, and how the application of constraint programming affects them. The second part (sections 6.3, 6.4 and 6.5) analyzes the impact of the order book oracle in the random trading simulations, by referring to the collected results.

6.2 Characteristics of the order book oracle

Table 2.1 lists the main characteristics that must be considered in the development of a test oracle. This section gives an overview of the developed order book oracle by showing and discussing results related to each characteristic listed in the table. A discussion on how the use of constraint programming affects some characteristics is given as well. The metrics given here (for example, the functional requirement coverage) should be, in general, taken as rough approximations rather than perfect measurements of each characteristic.

6.2.1 Completeness

The completeness of the order book oracle has been evaluated by two main coverage measures: the percentage of order attributes that are successfully modeled, and the percentage of functional requirements related to auto-matching that are reflected in the order book model.

Subsection 4.3.2 shows the main attributes of a trade order in *TRADExpress*. Although only 6 different attributes are shown, an actual trade order message contains a total of 29 attributes that affect the way in which the order is handled by the system. For each of these attributes, the percentage of possible values that the model can handle has been calculated. From these calculations, it has been derived that the order book oracle covers approximately a 45% of all possible values that the different attributes of a trade order can take.

Regarding requirement coverage, the order book model reflects 42 from a total of 172 requirements that have been considered relevant to the auto-matching functionality modeled in this project, that is approximately a 27% of them.

A rough and purely qualitative inference of the two figures shown above gives that the test oracle covers approximately one third of the auto-matching functionality. However, a qualitative analysis of the measurements (assessing the importance of each auto-matching feature) might lead to different conclusions, as many functional requirements and trade order attributes deal with functionality that is not commonly applied in *TRADExpress*' production. A complete analysis in this direction would require the development of an operational profile, which, as mentioned in subsection 5.4.4, falls out of the scope of this thesis.

6.2.2 Accuracy

There is only one known case where the output of the order book oracle might be wrong, and it is documented in subsection 5.7.2. As explained there, the false positive can only be obtained in quite limited situations. Table 6.4 below shows that this false failure has arisen 25 times in a total of 45144 input actions, which

gives a Mean Time Between Failures (MTBF, see subsection 2.4.2) of 1806 actions. The relatively high MTBF of this defect makes its impact on the oracle accuracy low, although it would be of course desirable to correct it.

The use of a declarative programming paradigm such as constraint programming impacts positively on the correctness and accuracy of the test oracle. Assuming that the applied constraint programming system is correct, the directness of the specification-to-code translation reduces the proof of correctness of the order match and re-match code to verifying the specification itself.

Other aspect related to the accuracy of a test oracle, as described in [18], is the degree of independence with respect to the system under test. The order book oracle has been developed based on the requirements database and other documents, without referring to the implementation of *TRADExpress*, and it has been designed using different programming paradigms (constraint programming) and languages (C++). The degree of independence of the order book oracle is, therefore, considerably high.

6.2.3 Usability

As subsection 5.6.1 mentions, the developed test oracle provides output traceability, event and statistics logging and possibility of repeating test cases. In addition, it is a highly portable software system, written in Java and standard C++ and using only libraries that are at the same level of portability. Other aspect that contributes to its usability (see [18]) is that the expected results are returned as matcher objects that can be directly used by the comparators available in the random test framework.

6.2.4 Maintainability

As section 2.3 discusses, a main concern in the design and implementation of a test oracle is to keep its complexity and maintainability as low as possible. One of the simplest software maintainability metrics is the number of lines of code (LOC) [37]. This measure has been calculated (discarding empty lines) for both the test oracle and the *TRADExpress* components that implement the functionality under test. Although the test oracle is developed in two different programming languages, empirical research shows that software systems implemented in Java and C++ are generally equivalent in LOC [29]. Table 6.1 shows the results, measured in January 2010 with the tool CLOC [1].

[1] CLOC v. 1.08. `http://cloc.sourceforge.net`

System	Files	Code lines	Comment lines
Test system	32	1946	902
TRADExpress	34	9585	4377

Table 6.1: LOC of the test oracle and the analogous *TRADExpress* components

As table 6.1 shows, the test oracle comprises approximately a 20% of the code and comment lines used in *TRADExpress* to deliver an equivalent functionality. The fact that almost the same number of code files have been used in both cases is explained by the use in the test oracle of C++ headers and one file for each Java comparator and predicate class. The important difference in LOC can be explained by many factors. The main ones are:

- *TRADExpress* implements the optimization search for the order re-match from scratch, while the test oracle reuses the search capabilities of the constraint programming system.

- Performance is a primary goal in the *TRADExpress*, while its importance in the test oracle is secondary (see subsection 6.2.5 below). As a result, the *TRADExpress* implementation contains more lines of code, for example, handling individual cases that could only be generalized with some lose of performance.

Furthermore, the closeness between the order matcher and re-matcher specifications and their implementation (due to the use of a a declarative programming paradigm) contributes positively to the maintainability of the test oracle.

6.2.5 Performance

The performance of the order book oracle is already identified in the introduction chapter as a secondary goal. The average execution time of the calculation of the expected trades (see algorithm 1) has been measured in the test runs described in section 6.3: while it is orders of magnitude higher (0.745 s. on average) than that of *TRADExpress* [5], the time bottleneck at the random test process is probably still the manual failure analysis. On the other hand, a main factor in this execution time is the maximum quantity that actors are allowed to trade with (see subsection 5.7.2). This parameter can be specified in the test scripts, so it can be easily reduced for a better execution time if necessary.

6.2.6 Cost

Besides the temporal costs discussed above, there are many other costs that must be considered in the development of an oracle. Some of the possible costs listed in [18], such as the maintenance cost of comparators, are cancelled out by the reutilization of the testing framework previously developed at Cinnober. The license cost of all technologies used in the test oracle implementation, including the constraint programming system, is null as well. The maintenance cost of the test oracle, considering the figures examined above, is low in comparison with the maintenance cost of the system under test.

6.3 Random test cases

6.3.1 The test cases

As chapter 5 states, the test cases that can be designed within the test system are highly configurable. This allows to create a practical infinite number of test cases that differ not only in the random generator seed number, but in other parameters such as the number of actors and action probabilities. For the sake of simplicity, a single configuration has been used and repeated with different random generator seed numbers. In order to guarantee a potentially full coverage of the input domain, all actions are included in the test configuration, distributed in three actors. The probabilities of actors and actions being chosen in a given simulation step are shown in table 6.2.

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

Table 6.2: Actors, actions and probabilities in the executed test cases

The test configuration is applied to 500 test cases with different generator seeds, of 100 actions each. Prices are generated in a range between 10 and 100, and quantities are generated in a range between 2 and 50. For the sake of simplicity and failure traceability, only one order book, with its corresponding oracle, is defined. While this input model does not aim to be specially realistic (as it is not based in an operational profile), it is designed to take advantage of all developed capabilities of the test oracle, and it can generate situations that are complex enough to detect low frequency defects in the system under test.

6.3.2 Results

The 500 test cases have been run during the days 13th and 14th of January 2010 against the version 8.0.0 of *TRADExpress*. Although a total of 50000 input actions were planned, premature terminations in the failing test cases have reduced the total input actions to 45144. Table 6.3 shows the total match and re-match hits calculated by the test oracle (see subsection 5.4.4) and their corresponding expected trades in the executed random tests.

Matching step	Total hits	Total trades
Match	10895	17037
Re-match	526	1189

Table 6.3: Total match and re-match hits and expected trades in the test runs

Two types of failure, explained with more detail in section 6.4, have been detected in the test execution:

- A pegged order wrongly remains in an empty order book.
- An incoming order without best priority is allowed to match.

Furthermore, the test oracle has erroneously reported several times a false re-match failure documented in subsection 5.7.2. Because the trading system is repairable for the found failures, in the sense that they do not cause any crash or functionality loss, it is possible to calculate the Mean Time Between Failures (MTBF, see subsection 2.4.2). Table 6.4 shows the calculated MTBF for each failure, including the false one, and the total MTBF.

Failure	Occurrences	MTBF (actions)
Pegged order in empty order book	51	885
Incoming low priority order matches	37	1220
Re-match false positive	25	1805
Total	113	399

Table 6.4: MTBF of each failure detected in the random test cases

6.4 Problems found in the trading system

6.4.1 System failures

Pegged order remains in an empty order book

After a re-match that fills all orders that a pegged order can peg to and partially fills a pegged order, the order book lacks a visible Market Best Bid and Offer price (MBBO, see section 4.4). Because of this, the pegged order is canceled as expected, and the order book is emptied. However, if the same sequence is repeated, the pegged order remains in the order book, even thought there is no order that can define a visible MBBO and it should be automatically cancelled by the system. The following steps are a simplification of the sequence causing the failure:

1. The order b_1, which is pegged to the best bid price with a price offset of -1, is inserted into an order book containing b_0 and a_0:

→ **Bid orders**	**Ask orders**
$b_0 : 13$ @ 47 $\rightarrow b_1 : 17$ @ 46	$a_0 : 14$ (≥ 14) @ 22

2. A match attempt is performed without success, and the following trades are calculated in the re-match:

$$b_0 \leftrightarrow a_0 : 13 \text{ @ } 22$$
$$b_1 \leftrightarrow a_0 : 1 \text{ @ } 22$$

   As a result, b_0 and a_0 are completely filled, and b_1 is automatically cancelled by the system, as there is no defined visible MBBO.

3. Step 1 is repeated.

4. The same trades as in step 2 are executed, but this time the pegged order b_1 wrongly remains in the order book:

Bid orders	**Ask orders**
b_1 : 16 @ ?	

Incoming order without best priority is allowed to match

As the common order matcher constraint problem defines, no order is allowed to match against orders in the other side if there are unfilled non-minimum quantity condition orders with better priority in its side. However, if a visible MBBO is defined (this is, if there are non-minimum quantity orders in both sides of the order book) an incoming order with the same limit price but worse priority than other orders existing in its side is wrongly allowed by *TRADExpress* to match, by-passing them. The following steps are a simplification of the sequence generated by the test system that causes the failure:

1. The AoN order a_1 is inserted in an order book containing b_0, a_0 and a_2. a_1 has the same limit price than a_0, but it gets worse priority for being a minimum quantity order:

Bid orders	$\rightarrow$ **Ask orders**
b_0 : 18 ($\geq$ 18) @ 32	a_0 : 4 @ 28 $\rightarrow$ a_1 : 18 ($\geq$ 18) @ 28 a_2 : 16 @ 38

2. The incoming order a_1 is wrongly allowed to match in the first order match attempt, even though a_0 has higher priority and, because it is not a minimum quantity order, it should never be bypassed by an order with lower priority:

$$b_0 \leftrightarrow a_1 : 18 \text{ @ } 32$$

6.4.2 Documentation defects

The development of the order book model has required a formalization of the *TRADExpress* order matching requirements, which were stated only in natural language at the beginning of this project. As a side benefit of this process, some documentation mistakes and ambiguities have been detected and fixed. Two main types of defects have been identified in a total of six corrected requirements:

- Ambiguous statements due to the use of a natural language (English).
- Erroneous calculations and system expectations in illustrative examples.

6.5 Impact of the order book oracle

As chapter 1 states, the main contribution of this work with respect to the previous work performed at Cinnober is the introduction of a true oracle, able to calculate the expected output of a *TRADExpress* order book for a subset, detailed in subsection 5.4.2, of its input domain.

In [19], a total of five defects are found in *TRADExpress* by applying random testing. Although an heuristic test oracle is developed, it actually does not take part in the detection of any of the found failures. In [21], no test oracle is used, and no failures are discovered. The author argues that "the system under test has been tested frequently and very rigidly for a long time, so no major errors were expected to be found" [21, p. 66]. Yet two new defects has been discovered in this project, even though *TRADExpress* is older and thus presumably more mature, due to the introduction of a true test oracle. This suggests the idea that the random testing capacity of finding new failures in *TRADExpress* without using a more complete test oracle was exhausted.

It is important to notice that a less complete test oracle than the true oracle could have given some results as well: a heuristic oracle verifying that an order book does not only contain pegged orders after each input action could have detected the remaining pegged order failure (see subsection 6.4.1). The incoming order failure is, however, harder to discover: detecting if an incoming order should match or not is equivalent in complexity to solving the match problem, which falls into the scope of a true oracle.

Finally, even if the use of a less complete test oracle could have helped to find failures, it is arguable whether many side benefits of the modeling and formalization task performed in this project, such as the documentation corrections described in subsection 6.4.2, would have been obtained.

Chapter 7

Conclusion and further work

7.1 Conclusion

This thesis aimed to provide a solution to the test oracle problem raised in the previous work on random testing applied to the trading system *TRADExpress*. The proposed solution has been to develop a model of the core component of *TRADExpress*, the order book, by applying constraint programming, and to integrate the model as a true oracle into the existing random test framework. This work has involved answering, through empirical research, two main questions:

1. In which degree does constraint programming represent a satisfactory solution for modeling complex existing systems such as *TRADExpress*?
2. How does the testing process improve by the addition of a true test oracle to the random testing framework?

The following two subsections summarize the work related to each question and draw the respective conclusions.

7.1.1 Constraint programming and the order book model

The main procedure executed in the order book model applies two different continuous double auctions (called *match* and *re-match*). The specifications of these auctions, initially given as functional requirements stated in a natural language, have been formalized into constraint satisfaction problems to be solved by a constraint programming system. This has been the first attempt, as far as the author is aware, to model a non-theoretical continuous double auction problem with this technique.

Because the model has been integrated as a true test oracle, its main design goals are those of the oracle: completeness, accuracy, and maintainability. Constraint programming has proven to support well these goals: because of the use of a different programming paradigm to that of *TRADExpress*, the test oracle is highly independent. Furthermore, constraint programming, as a declarative programming paradigm, has contributed to its accuracy by narrowing the gap between specifications and implementation. Last, the reuse of the search capabilities of the constraint programming system and the high level of abstraction of the oracle implementation have reduced its number of lines of code to a fifth of those used in *TRADExpress* to deliver the same functionality, contributing to a lower complexity and a higher maintainability.

From this discussion, the first question can be answered: constraint programming has proven to be suitable for modeling a complex existing system comprising combinatorial problems, and has shown to contribute well to the common design goals followed in the development of test oracles.

7.1.2 The order book model as a true test oracle

The second step in the proposed solution has been to integrate the order book model as a test oracle into the existing random testing framework. The test oracle updates the state of the order book model and triggers the computation of the double auctions through different actions, such as insertions and cancellations of trade orders. An actor that exercises approximately a third of the functionality of the main matching mechanism, including the most typical trade order types, has been added to the framework. Finally, a set of random test cases with the new configuration has been designed and run.

In 500 executed test cases, with a total of 45144 input actions, the true test oracle has detected two failures that had been present in *TRADExpress* for several years. These failures had not been discovered by previous random testing approaches with less complete test oracles. While one of these failures might have actually been detected with a less complete test oracle, detecting the other one would have been hard without a true oracle. Furthermore, the development of an independent, well-defined model of the system as a solution to the oracle problem has implied a formalization task that has brought side benefits, such as the disambiguation and correction of functional requirements.

Following this discussion, an answer can be given to the second question: a true test oracle has shown to significantly improve the capacity of finding failures through random testing. Although the development of a true test oracle based on an independent model of the system can be costly, the application of suitable declarative programming techniques can lower a great part of the implementation effort, besides bringing some side benefits derived from their reduction of the gap

between specification and implementation.

7.2 Further work

This thesis can be seen as a continuation of the work done in [19] and [21] on random testing in the context of the trading system *TRADExpress*. The obtained results encourage further research in the directions followed in this project.

7.2.1 Further development of the system model

The developed test oracle covers approximately a third of the functionality of the main matching mechanism offered by *TRADExpress*. An obvious continuation of this work is to enlarge the model with more order types and order book configurations, with the goal of increasing the test coverage and finding new failures. Other matching mechanisms offered by *TRADExpress*, like the so-called *auction*, could be as well modeled and tested in a similar way as it is done here.

7.2.2 Continuous double auction as a dynamic CSP

In a continuous double auction as the one performed in the *auto-matching* mechanism, every trade order action triggers a matching computation that typically alters the order book in a limited way. Instead of building a new constraint satisfaction problem (CSP) and solving it from scratch after each user action (as it has been done in this project), the problem could be more naturally modeled as a dynamic CSP [26]. In this way, the match computation could be made more effective, as the results of the previous computations would be reused.

7.2.3 Automatic test case reduction

To manually isolate a failure that has been found in a long random test case is a very tedious task. This process can fortunately be automated (see, for example, [36, Ch. 13]). The random testing framework could be adapted by following the steps suggested in subsection 5.4.4, and the benefits of automatic test case reduction could be measured by, for example, inserting artificial failures in the system under test and calculating the test case reduction rate in several situations.

7.2.4 Operational profiles and reliability assessment

Reliability assessment is often included in the list of goals of software testing. As section 2.4 points out, reliability estimation requires the use of a realistic operational profile in the test input distribution. The development of an operational

profile, following for example the methods detailed in [27], based on real usage data of *TRADExpress* could be an interesting step towards the creation of realistic reliability estimation models such as the ones described in subsection 2.4.2.

References

[1] Hakim Belhaouari and Frédéric Peschanski. A constraint logic programming approach to automated testing. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 754–758, Berlin, Heidelberg, 2008. Springer-Verlag. Cited on page 5.

[2] Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, 2002. Cited on page 5.

[3] Cinnober Financial Technology AB. *TRADExpress Trading Engine - Functional Requirements Database*, 2009. Cited on pages 28, 30, 31, 32, and 34.

[4] Cinnober Financial Technology AB. *TRADExpress Trading Engine - System Architecture Course*, 2009. Cited on pages 27 and 61.

[5] Cinnober Financial Technology AB. *TRADExpress Trading System - System Overview*, 2009. Cited on pages 1, 2, 25, 26, and 66.

[6] D. Epstein. On the NP-completeness of cryptarithms. *SIGACT News*, 18(3):38–40, 1987. Cited on page 17.

[7] Ming Fan, Jan Stallaert, and Andrew B. Whinston. The internet and the future of financial markets. *Commun. ACM*, 43(11):82–88, 2000. Cited on page 1.

[8] Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997. Cited on pages 4 and 15.

[9] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, London, UK, 1995. Springer-Verlag. Cited on page 3.

[10] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998. Cited on page 5.

[11] Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Int. Thomson Business Press, 2006. Cited on pages 3, 7, 9, 12, and 13.

[12] Johan Gundemark. Random tests in a market place system. Master's thesis, Uppsala University, Uppsala, Sweden, 2005. Cited on pages 2, 5, and 9.

[13] Dick Hamlet. When only random testing will do. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 1–9, New York, USA, 2006. ACM. Cited on pages 8 and 9.

[14] Joel Hasbrouck. *Empirical Market Microstructure: The Institutions, Economics, and Econometrics of Securities Trading*. Oxford University Press, New York, USA, 2007. Cited on pages 6 and 31.

[15] Douglas Hoffman. A taxonomy for test oracles. *Proceedings of 11th International Quality Week*, 1998. Cited on pages 10 and 11.

[16] Douglas Hoffman. Heuristic test oracles. *Software Testing and Quality Engineering*, pages 29–32, March/April 1999. Cited on page 9.

[17] Douglas Hoffman. Mutating automated tests. *StarEast*, 2000. Cited on page 8.

[18] Douglas Hoffman. Using oracles in test automation. *Proceedings of Pacific Northwest Software Quality Conference*, pages 90–117, 2001. Cited on pages 8, 10, 65, and 67.

[19] Noah Höjeberg. 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, Stockholm, Sweden, 2008. Cited on pages 2, 3, 10, 13, 71, and 75.

[20] Jayant R. Kalagnanam, Andrew J. Davenport, and Ho S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. *Electronic Commerce Research*, 1(3):221–238, 2001. Cited on pages 6 and 31.

[21] David Karlgren. Random testing of a market place system. Master's thesis, Division of Mathematical Statistics, KTH Royal Institute of Technology, Stockholm, Sweden, 2009. Cited on pages 2, 3, 71, and 75.

[22] Mikael Zayenz Lagerkvist. Techniques for efficient constraint propagation, 2008. Cited on pages 15 and 20.

[23] H. G. Lee and R. M Lee. Electronic call market for commodity transactions: Design of computer-mediated order matching system. *Journal of Organizational Computing and Electronic Commerce*, 8(4):307–334, January 1999. Cited on page 6.

[24] Andreas Lundgren. Abstraction levels of automated test scripts. Master's thesis, Faculty of Engineering, Lund University, Lund, Sweden, 2008. Cited on page 2.

[25] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2):81–96, 2001. Cited on page 5.

[26] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 25–32, 1990. Cited on page 75.

[27] J.D. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14–32, Mar 1993. Cited on pages 9 and 76.

[28] M. Petit and A. Gotlieb. An ongoing work on statistical structural testing via probabilistic concurrent constraint programming. In *First International Workshop on Model, Design and Validation. Proceedings*, pages 19–27, Nov. 2004. Cited on page 5.

[29] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct 2000. Cited on page 65.

[30] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Inc., New York, USA, 2006. Cited on pages 4, 15, 16, 19, 21, 22, and 23.

[31] Young U. Ryu. Hierarchical constraint satisfaction of multilateral trade matching in commodity auction markets. *Annals of Operations Research*, 71(0):317–334, January 1997. Cited on page 6.

[32] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling with Gecode*, 2009. Cited on page 59.

[33] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. Doctoral Dissertation, Saarland University, Jan 2009. Cited on page 15.

[34] Georg von Zweigbergk. An introduction to financial markets. Technical report, Cinnober Financial Technology AB, 2009. Cited on pages 1, 25, 26, 28, 29, 30, and 32.

[35] M. Xie. *Software reliability modelling*. World Scientific, 1991. Cited on page 13.

[36] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005. Cited on pages 13 and 75.

[37] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1997. Cited on page 65.

Glossary

auto-matching: matching mechanism of a trading system where trade orders can be inserted and executed on a continuous basis.

automated testing: testing which is supported or performed by a computer.

black-box testing: testing without reference to the internal structure of the system.

bug: see *defect*.

complete test oracle: see *true test oracle*.

constraint problem: see *constraint satisfaction problem*.

constraint programming: programming paradigm where constraints over variables are defined, and a general purpose constraint system is used to solve them.

constraint propagation: process applied in constraint programming to discard values that are not part of the solution of a problem.

constraint satisfaction problem (CSP): mathematical problem defined as a set of variables with finite domains and a finite set of constraints.

constraint: relation over the variables of a problem that must be satisfied by the problem solutions.

cost function: function that quantifies how good a particular solution to a problem is.

debugging: process of looking for the defect that causes a certain failure in a system.

declarative programming: programming paradigm where *what* should be computed is described, rather than *how* the computation should be performed.

defect: a flaw in a system that may cause the system to fail to perform its required function.

error: a human action that produces an incorrect result.

failure rate: number of expected failures per time unit in a system.

failure: deviation of a system from its expected behavior.

financial instrument: item of value, physical or not, on which trades can be performed.

financial market: system that allows buyers and sellers to trade and exchange on financial instruments.

functional requirement: a requirement that specifies a function that a component or system must perform.

functional testing: testing that analyzes the specification of the system functionality.

market participant: trader who takes part in the exchange powered by a financial market.

match: first matching step applied in *auto-matching* after a trade order is inserted or updated.

mean time between failures (MTBF): average time between failures of a system.

NP-complete: a class of problems for which no polynomial-time algorithms are known.

operational profile: quantitative characterization of how a system will be used.

order book: component of a trading system where different trade orders referring to a certain instrument are stored and matched.

random testing: a testing technique where test cases are selected randomly to match an operational profile.

random trading simulation: computer-based simulation that imitates a trading sequence in a financial market, where different participants introduce random trade orders matching a certain operational profile.

re-match: second matching step applied in *auto-matching* after a trade order is inserted or updated. It is also applied immediately after an order cancellation.

regression testing: testing of a previously tested system that has been modified to ensure that defects have not been introduced during the modification.

software reliability: ability of a software system to perform its required functions for a specified period of time or number of operations.

software testing: process of both detecting defects in a software system and determining if it meets the requirements and fits its purpose.

systematic testing: as opposite to random testing, testing where the most valuable test cases are selected by applying techniques such as equivalence partitioning.

test case: a set of input values and expected results developed for a particular objective.

test oracle: a source to determine expected results to compare with the actual result of the system under test.

trade order: request of buying or selling a certain quantity of a financial instrument on a specific financial market.

trade: transaction that involves the exchange of a financial instrument for a certain amount of money.

trading system: software system that manages the execution of trade orders in a financial market. Cinnober's *TRADExpress* is an example of a trading system.

true test oracle: independent implementation of the system under test that reproduces the output that is expected from it. See *test oracle*.

Appendix A

Java code documentation

A.1 OrderBookActor

A.1.1 Fields

public static final String INSERT_LIMIT_ORDER

public static final String INSERT_FILL_AND_KILL_ORDER

public static final String INSERT_FILL_OR_KILL_ORDER

public static final String INSERT_MARKET_ORDER

public static final String INSERT_PEGGED_ORDER

public static final String INSERT_ALL_OR_NONE_ORDER

public static final String UPDATE_ORDER

public static final String CANCEL_ORDER

A.1.2 Constructors

public OrderBookActor(Simulation pSimulation, String pRoleName, String pUserAlias, Instrument pInstrument, int pWeight)

Initializes the order book actor.

A.1.3 Methods

public final void act(long pCycle)

Cleans the public trade flow message queue and picks and performs an action.

public OrderBookActor useOrderBookOracle(OrderBookOracle pOrderBookOracle)

Specifies that the actor will use pOrderBookOracle for verifying the public trade flow.

public OrderBookActor setQtyLimits(int pMinQty, int pMaxQty)

Sets the lower (pMinQty) and higher (pMaxQty) limits for quantity generation.

public OrderBookActor setPriceLimits(int pMinPrice, int pMaxPrice)

Sets the lower (pMinPrice) and higher (pMaxPrice) limits for price generation.

public OrderBookActor setPriceDecimalLevel(int pPriceDecimalLevel)

Specifies the number of decimal digits that the actor will use in its order prices. Note that a high decimal level can lead to round-off errors and thus incorrect expected trade prices.

public int getPriceDecimalLevel()

Returns the number of decimal digits that the actor will use in its order prices.

public void insertLimitOrder()

Inserts a limit order.

public void insertMarketOrder()

Inserts a market order.

public void insertFillAndKillOrder()

Inserts a fill-and-kill order.

public void insertFillOrKillOrder()

Inserts a fill-or-kill order.

public void insertPeggedOrder()

Inserts a pegged order.

public void insertAllOrNoneOrder()

Inserts an all-or-none order.

public void updateOrder()

Updates an order.

public void cancelOrder()

Cancels an order.

public void restoreState()

Cancels all its orders in the order book.

A.2 OrderBookOracle

A.2.1 Constructors

public OrderBookOracle(Simulation pSimulation, String pUserAlias, Instrument pInstrument)

Initializes the order book oracle.

A.2.2 Methods

public void prediction()

Prepares the EMAPI subscriptions for all the participating actors.

public Instrument getInstrument()

Returns the tradable instrument the oracle looks at.

public void addEmapiActor(OrderBookActor pOrderBookActor)

Adds pOrderBookActor to the list of actors that participate in the trading for the order book.

public void cleanOrderEventQueues()

Cleans the order event message queues of all actors.

public void printOrderBook()

Prints out the oracle's internal order book state.

public EmapiTradeEventMatcher[] getExpectedTradeEvents()

Returns an array of matchers for the trade events that are expected to happen as a consequence of the last order action performed in the order book.

public List getAllPrivateOrderEvents()

Returns all private order events registered in the internal order book.

public long getBestLimitPriceInSide(boolean pIsBid)

Returns the best limit price in the given side of the internal order book. If there are no orders in the side, it returns 0.

public void restoreState()

Restores the state of the order book.

A.3 OrderBookModel

A.3.1 Constructors

public OrderBookModel(Instrument pInstrument, double pTick)

Initializes the order book model.

A.3.2 Methods

public void registerOrderEvent(EmapiOrderEventPrivateIf pOrderEvent)

Updates the order book model with the order event pOrderEvent.

public long getBestLimitPriceInSide(boolean pIsBid)

Returns the best limit price in the given side of the order book. If there are no orders in the side, it returns 0.

public List getOrderEventList()

Returns a list with all private order events in the order book.

public List match()

Returns a list of trade event matchers that are expected to be the result of matching the contained orders. Calls the external library "ordermatcher".

public List rematch(boolean pIsBidTheIncomingSide)

Returns a list of trade event matchers that are expected to be the result of applying a re-match to the contained orders. Calls the external library "ordermatcher".

public List toTradeEventMatcherList(List pTrades)

Converts a list of TradeModel objects to a list of trade event matcher objects.

public List getCancelOrderEventMatchersAfterMatch()

Returns a list of cancel order event matchers with the cancel events expected after a match attempt for those orders that are valid only until the next match.

public List getOrderEventMatchersFromTrades(List pTrades)

Returns a list of order event matchers that are expected to be caused by the trades in pTrades.

public List getPriceRecalculationOrderEventMatchers()

Returns a list of order matchers due to price recalculation of pegged orders.

public List getCancelPeggedOrderEventMatchers(List pBaseOrders)

Returns a list of order matchers due to cancel of pegged orders when there is nothing to peg to. It includes only the order events that have not occurred yet in pBaseOrders.

public String toString()

Returns a string with a representation of the order book state.

Appendix B

Test cases for the order matching

As section 5.7 explains, the trade calculation subsystem, which includes the classes *OrderMatcherBaseSpace*, *OrderMatcherSpace* and *OrderRematcherSpace*, has been developed by following a test-driven development style: on top of a test framework based on CppUnit (a C++ testing framework), automated test cases defining the expected behavior of the order matcher and re-matcher have been added, and then the functionality has been implemented and verified by just running the test cases. This has been performed iteratively, according to the cycles illustrated in figure 5.7.

The test suite includes a total of 77 test cases: 46 of them test the order matcher, and the remaining 31 test the order re-matcher. For the sake of conciseness, only those test cases that are considered to be more illustrative are included in this appendix. Section B.1 contains the selected test cases for the order matcher, and section B.2 contains the selected test cases for the order re-matcher. All test cases follow the notation introduced in chapter 4. Each test case includes a title, an optional description, a figure with the input order book state, and the expected output.

B.1 Test cases for the order matcher

B.1.1 Test cases with non-minimum quantity orders only

1. Incoming order completely fills resting order

- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 25 \ @ \ 100$	$\rightarrow a_0 : 25 \ @ \ 100$

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 25 \ @ \ 100$$

2. **Incoming order completely fills several resting orders**

- **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 100 \ @ \ 100$	$a_0 : 25 \ @ \ 100$ $a_1 : 75 \ @ \ 100$

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 25 \ @ \ 100$$
$$b_0 \leftrightarrow a_1 : 75 \ @ \ 100$$

3. **Incoming order gets highest priority in its side**

- **Description:** The incoming order gets higher priority than a_1 because of its better price, and is allowed thus to completely fill the two resting bid orders.

- **Input:**

Bid orders	→ **Ask orders**
$b_0 : 25 \ @ \ 100$ $b_1 : 35 \ @ \ 100$	→ $a_0 : 75 \ @ \ 100$ $a_1 : 150 \ @ \ 105$

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 25 \ @ \ 100$$
$$b_1 \leftrightarrow a_0 : 35 \ @ \ 100$$

4. **Incoming order partially fills the resting order with highest priority**

- **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 30 \ @ \ 25$	$a_0 : 200 \ @ \ 23$ $a_1 : 30 \ @ \ 24$ $a_2 : 10 \ @ \ 25$

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 30 \ @ \ 23$$

B.1.2 Test cases with incoming minimum quantity orders

1. **Incoming minimum quantity order does not reach match limit price**

   - **Description:** Even though b_0 gets the highest priority in its side, the best ask order (which offers exactly the quantity needed) cannot trade at a price lower than 101, so no match is possible.
   - **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 100\ (\geq 100)$ @ 100 $b_1 : 100$ @ 99	$a_0 : 100$ @ 101

   - **Expected output:** (no trades expected)

2. **Incoming minimum quantity order cannot fill its minimum quantity**

   - **Description:** The incoming order a_0 is only allowed to match b_0, because of the price constraints. However, b_0 cannot fill it completely, so no match is possible.
   - **Input:**

Bid orders	→ **Ask orders**
$b_0 : 2000$ @ 229 $b_1 : 3000$ @ 228	→ $a_0 : 3000\ (\geq 3000)$ @ 229

   - **Expected output:** (no trades expected)

3. **Incoming minimum quantity order is completely filled**

   - **Description:** The incoming order a_0 can match against both b_0 and b_1. These two orders can contribute with the required quantity, so two trades are created in priority order, at the price set by the orders in the resting side.
   - **Input:**

Bid orders	→ **Ask orders**
$b_0 : 2000$ @ 229 $b_1 : 3000$ @ 228	→ $a_0 : 3000\ (\geq 3000)$ @ 228

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 2000 \text{ @ } 229$$
$$b_1 \leftrightarrow a_0 : 1000 \text{ @ } 228$$

B.1.3 Test cases with resting minimum quantity orders

1. **Incoming order is completely filled**

   - **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 40$ @ 6	$a_0 : 20$ (≥ 20) @ 5 $a_1 : 30$ @ 6

   - **Expected output:**

$$b_0 \leftrightarrow a_0 : 20 \text{ @ } 5$$
$$b_0 \leftrightarrow a_1 : 20 \text{ @ } 6$$

2. **Incoming order is completely filled**

   - **Description:** The price of the trade between b_0 and a_0 is moved to the best possible visible price existing in the order book before b_0 is inserted.

   - **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 40$ @ 60 $b_1 : 10$ @ 55	$a_0 : 20$ (≥ 20) @ 50 $a_1 : 30$ @ 60

   - **Expected output:**

$$b_0 \leftrightarrow a_0 : 20 \text{ @ } 55$$
$$b_0 \leftrightarrow a_1 : 20 \text{ @ } 60$$

3. **Incoming order bypasses a_0 and is completely filled**

   - **Description:** The minimum quantity constraint of a_0 cannot be satisfied. Minimum quantity orders can be bypassed, so b_0 matches against the lower priority limit order a_1.

   - **Input:**

→ **Bid orders**	**Ask orders**
→ $b_0 : 40$ @ 6	$a_0 : 50$ (≥ 50) @ 5 $a_1 : 30$ @ 6

- **Expected output:**

$$b_0 \leftrightarrow a_1 : 30 \text{ @ } 6$$

4. **Incoming order cannot fill alone a resting order**

   - **Description:** In order to satisfy the minimum quantity constraint of a_0, b_1 needs the quantity of b_0. This situation can only be solved by executing a re-match.

   - **Input:**

→ **Bid orders**	**Ask orders**
$b_0 : 100$ @ 104 → $b_1 : 100$ (≥ 100) @ 104	$a_0 : 200$ (≥ 200) @ 103

   - **Expected output:** (no trades expected)

5. **Incoming order is filled against several orders**

   - **Input:**

Bid orders	→ **Ask orders**
$b_0 : 10$ @ 100 $b_1 : 10$ (≥ 10) @ 100 $b_2 : 5$ (≥ 5) @ 100 $b_3 : 3$ @ 99	→ $a_0 : 18$ @ 98

   - **Expected output:**

$$b_0 \leftrightarrow a_0 : 10 \text{ @ } 100$$
$$b_2 \leftrightarrow a_0 : 5 \text{ @ } 100$$
$$b_3 \leftrightarrow a_0 : 3 \text{ @ } 99$$

6. **Incoming order is filled against best resting order**

   - **Description:** The price of the trade between b_0 and a_0 is moved to the best possible visible price existing in the order book before a_0 is inserted.

- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 1000\ (\geq 1000)\ @\ 1210$	$\rightarrow a_0 : 1000\ @\ 1207$
$b_1 : 2000\ @\ 1207$	$a_1 : 200\ @\ 1208$

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 1000\ @\ 1208$$

7. Incoming order fills best resting order

- **Description:** Orders with minimum quantity constraints can be bypassed by other orders. Therefore, although a_1 has a worse price than a_0, it gets the possibility of matching b_0.
- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 3\ (\geq 3)\ @\ 646$	$a_0 : 9\ (\geq 9)\ @\ 187$
	$\rightarrow a_1 : 10\ @\ 417$

- **Expected output:**

$$b_0 \leftrightarrow a_1 : 3\ @\ 646$$

8. Incoming order cannot bypass resting order

- **Description:** Only orders with minimum quantity constraints can be bypassed by other orders. As b_0 does not have any minimum quantity constraint, the incoming order b_1 is not allowed to match.
- **Input:**

→ Bid orders	**Ask orders**
$b_0 : 1\ @\ 382$	$a_0 : 9\ (\geq 9)\ @\ 313$
$\rightarrow b_1 : 10\ @\ 377$	$a_1 : 10\ (\geq 10)\ @\ 458$

- **Expected output:** (no trades expected)

B.2 Test cases for the order re-matcher

B.2.1 Simple test cases

1. **Simple situation where all quantity is filled**

   - **Input:**

Bid orders	**→ Ask orders**
$b_0 : 100\ (\geq 100)\ @\ 100$	$a_0 : 25\ @\ 100$ $a_1 : 75\ @\ 100$

   - **Expected output:**

$$b_0 \leftrightarrow a_0 : 25\ @\ 100$$
$$b_0 \leftrightarrow a_1 : 75\ @\ 100$$

2. **Minimum quantity order cannot bypass limit order**

   - **Description:** For b_1 to be allowed to trade b_0 should be completely filled, but this cannot happen without violating the minimum quantity constraint of b_1, so no trade is possible.

   - **Input:**

→ Bid orders	**Ask orders**
$b_0 : 100\ @\ 106$ $b_1 : 100\ (\geq 100)\ @\ 106$	$a_0 : 150\ (\geq 150)\ @\ 104$

   - **Expected output:** (no trades expected)

3. **Minimum quantity order bypassed to trade**

   - **Description:** Although a_0 has the highest priority in its side, it is bypassed by the rest of ask orders in order to satisfy the minimum quantity constraint of b_0 and trade.

   - **Input:**

→ Bid orders	**Ask orders**
$b_0 : 9000\ (\geq 9000)\ @\ 134$	$a_0 : 1000\ (\geq 1000)\ @\ 131$ $a_1 : 2000\ (\geq 2000)\ @\ 132$ $a_2 : 3000\ (\geq 3000)\ @\ 133$ $a_3 : 4000\ (\geq 4000)\ @\ 134$

- **Expected output:**

$$b_0 \leftrightarrow a_1 : 2000 \ @ \ 132$$
$$b_0 \leftrightarrow a_2 : 3000 \ @ \ 133$$
$$b_0 \leftrightarrow a_3 : 4000 \ @ \ 134$$

4. **Trades are executed in priority order**

- **Description:** The trades are distributed in a way such that the orders with highest priority in each side get to trade with each other.
- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 3 \ (\geq 3) \ @ \ 100$	$a_0 : 7 \ (\geq 7) \ @ \ 100$
$b_1 : 3 \ (\geq 3) \ @ \ 100$	$a_1 : 7 \ (\geq 7) \ @ \ 100$
$b_2 : 3 \ (\geq 3) \ @ \ 100$	$a_2 : 7 \ (\geq 7) \ @ \ 100$
$b_3 : 3 \ (\geq 3) \ @ \ 100$	
$b_4 : 3 \ (\geq 3) \ @ \ 100$	
$b_5 : 3 \ (\geq 3) \ @ \ 100$	
$b_6 : 3 \ (\geq 3) \ @ \ 100$	

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 3 \ @ \ 100$$
$$b_1 \leftrightarrow a_0 : 3 \ @ \ 100$$
$$b_2 \leftrightarrow a_0 : 1 \ @ \ 100$$
$$b_2 \leftrightarrow a_1 : 2 \ @ \ 100$$
$$b_3 \leftrightarrow a_1 : 3 \ @ \ 100$$
$$b_4 \leftrightarrow a_1 : 2 \ @ \ 100$$
$$b_4 \leftrightarrow a_2 : 1 \ @ \ 100$$
$$b_5 \leftrightarrow a_2 : 3 \ @ \ 100$$
$$b_6 \leftrightarrow a_2 : 3 \ @ \ 100$$

5. **Minimum quantity order is bypassed by limit orders**

- **Description:** The minimum quantity order b_1 does not belong to any possible combination. Therefore, it is bypassed by b_2 and b_3, which fill completely (together with b_0) the orders in the ask side.

- **Input:**

→ **Bid orders**	**Ask orders**
$b_0 : 10 \ @ \ 10$	$a_0 : 15 \ (\geq 15) \ @ \ 10$
$b_1 : 100 \ (\geq 100) \ @ \ 10$	$a_1 : 15 \ (\geq 15) \ @ \ 10$
$b_2 : 10 \ @ \ 10$	
$b_3 : 10 \ @ \ 10$	

- **Expected output:**

$$b_0 \leftrightarrow a_0 : 10 \ @ \ 10$$
$$b_2 \leftrightarrow a_0 : 5 \ @ \ 10$$
$$b_2 \leftrightarrow a_1 : 5 \ @ \ 10$$
$$b_3 \leftrightarrow a_1 : 10 \ @ \ 10$$

6. **Undefined equilibrium price prevents trading**

- **Description:** All possible subsets of orders in both sides that could satisfy their respective minimum quantity constraints and trade (for example, $\{b_0, b_4\}$ and $\{a_0, a_1\}$) are found at different equilibrium price levels, so it is not possible to find an equilibrium price that allows trading.

- **Input:**

→ **Bid orders**	**Ask orders**
$b_0 : 14 \ (\geq 14) \ @ \ 995$	$a_0 : 20 \ (\geq 20) \ @ \ 209$
$b_1 : 31 \ (\geq 31) \ @ \ 827$	$a_1 : 5 \ (\geq 5) \ @ \ 565$
$b_2 : 40 \ (\geq 40) \ @ \ 675$	
$b_3 : 13 \ (\geq 13) \ @ \ 631$	
$b_4 : 11 \ (\geq 11) \ @ \ 473$	
$b_5 : 11 \ (\geq 11) \ @ \ 275$	

- **Expected output:** (no trades expected)

B.2.2 More complex test cases

1. **Smaller order subsets are preferred for trading**

- **Description:** There are several possible combinations of orders that maximize the total traded quantity (for example, $\{b_1, b_6\}$ and $\{a_0, a_1\}$) and lie inside a defined equilibrium price. Among them, the smaller bid subset $\{b_5\}$ has higher preference, and its selected for filling $\{a_0, a_1\}$ with a total traded quantity of 40.

- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 27\ (\geq 27)\ @\ 19$	$a_0 : 5\ (\geq 5)\ @\ 11$
$b_1 : 20\ (\geq 20)\ @\ 15$	$a_1 : 35\ (\geq 35)\ @\ 11$
$b_2 : 9\ (\geq 9)\ @\ 14$	$a_2 : 28\ (\geq 28)\ @\ 14$
$b_3 : 24\ (\geq 24)\ @\ 13$	$a_3 : 23\ (\geq 23)\ @\ 15$
$b_4 : 36\ (\geq 36)\ @\ 11$	$a_4 : 31\ (\geq 31)\ @\ 17$
$b_5 : 40\ (\geq 40)\ @\ 11$	$a_5 : 30\ (\geq 30)\ @\ 18$
$b_6 : 20\ (\geq 20)\ @\ 11$	$a_6 : 28\ (\geq 28)\ @\ 18$
	$a_7 : 10\ (\geq 10)\ @\ 19$
	$a_8 : 5\ (\geq 5)\ @\ 19$
	$a_9 : 40\ (\geq 40)\ @\ 19$

- **Expected output:**

$$b_5 \leftrightarrow a_0 : 5\ @\ 11$$
$$b_5 \leftrightarrow a_1 : 35\ @\ 11$$

2. Must trade quantity must be always filled

- **Description:** There are several possible combinations of two ask orders that can fill completely b_0 (for example, $\{a_1, a_3\}$) and lie inside a defined equilibrium price. All of them need to include a_3. If the equilibrium price is set to 16, a_3 becomes a can trade order, and the absolute imbalance is not minimized as it cannot be completely filled. At an equilibrium price higher than 16, a_3 becomes a must trade order and it must necessarily be filled, as it is when is combined with a_4.

- **Input:**

Bid orders	**→ Ask orders**
$b_0 : 10\ (\geq 10)\ @\ 19$	$a_0 : 3\ (\geq 3)\ @\ 11$
	$a_1 : 6\ (\geq 6)\ @\ 12$
	$a_2 : 9\ (\geq 9)\ @\ 12$
	$a_3 : 5\ @\ 16$
	$a_4 : 5\ (\geq 5)\ @\ 17$
	$a_5 : 3\ (\geq 3)\ @\ 19$

- **Expected output:**

$$b_0 \leftrightarrow a_3 : 5\ @\ 19$$
$$b_0 \leftrightarrow a_4 : 5\ @\ 19$$