KTH Computer Science
and Communication

Random Tests in a Trading System

Random Tests in a Trading System Using Simulations and a Test Oracle

NOAH HÖJEBERG

Master's Thesis at NADA
Supervisor Cinnober: Lars Wahlberg
Supervisor KTH: Karl Meinke
Examiner: Stefan Arnborg

TRITA xxx yyyy-nn

Abstract

A large complex system, such as a trading system, is difficult to test. Regular automatic and manual tests can be very useful in finding errors but many bugs may go undetected. Random testing is a powerful tool in finding low-frequency bugs that are nearly impossible to find using other methods. Buy using randomly generated data in trading simulations, low-frequency bugs that may cause the system to crash, can be found and removed. By using a test oracle to verify the results, one may also detect non-crashing errors in the system.

This master's thesis focuses on random tests in general and on random tests in a trading system in particular, giving a basic explanation of the theory behind random tests and a practical study of a random test based on trading simulations and a test oracle.

Referat

Randomtester av handelssystem
Randomtester i ett börssystem med hjälp av simulationer och ett testorakel

Ett så stort complext system som ett börssystem är svårt att testa. Vanliga automatiska och manuella tester kan vara användbara till att hitta fel men många buggar kan missas. Randomtester är ett kraftfullt verktyg för att hitta lågfrekventa fel, som är så gott som omöjliga att hitta med andra metoder. Genom att använda slumpmässigt genererad data i handelssimulationer, kan man hitta och avlägsna lågfrekventa fel, som annars skulle kunna krascha systemet. Genom att använda ett orakel kan man även hitta fel som inte kraschar systemet.

Det här examensarbetet fokuserar på randomtester i allmänhet och randomtester i ett börssystem i synnerhet. Rapporten ger en grundläggande förklaring till teorin bakom random tester och en praktisk studie av ett randomtest baserad på handelssimulationer och ett testorakel.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem Description . . . 1
1.3 Overview . . . 2
1.4 Acknowledgments . . . 2

2 The Market Place System **3**
2.1 Market Place Actors . . . 3
2.1.1 Customers . . . 3
2.1.2 Brokers . . . 3
2.1.3 Market Makers and Specialists . . . 3
2.2 Instruments . . . 4
2.3 Orders and Order Books . . . 4
2.3.1 Orders . . . 4
2.3.2 Order Matching . . . 6
2.3.3 Order books . . . 6
2.3.4 Transparency . . . 6
2.4 The Trading System . . . 6
2.4.1 Trading States . . . 6
2.4.2 Order Book Rule Groups . . . 6
2.4.3 Orders . . . 7
2.4.4 Execution of Transactions . . . 7
2.4.5 Market Management . . . 8
2.4.6 The System's Components . . . 8

3 Software Testing **11**
3.1 Static Testing . . . 11
3.2 Dynamic Testing . . . 13
3.2.1 Black Box Testing . . . 13
3.2.2 White Box Testing . . . 14
3.2.3 White Box Analysis . . . 14
3.3 Experience-Based Testing . . . 15
3.3.1 Exploratory Testing . . . 15

3.3.2 Error Guessing 15
3.4 Automatic Testing 15

4 Random Tests 17
4.1 Introduction - Concept of Random Tests 17
4.2 Data Generation / Input 18
4.2.1 Distribution 18
4.2.2 Domain 18
4.3 Simulations 18
4.3.1 Trading Simulations 18
4.3.2 Modeling Actors/Human traders 19
4.4 Oracles 20
4.4.1 No Oracle 20
4.4.2 True Oracle 21
4.4.3 Consistent Oracle 21
4.4.4 Self-Referential Oracle 21
4.4.5 Heuristic Oracle 21
4.5 Reliability 22
4.5.1 Models for Software Reliability 22
4.5.2 Low-Frequency Errors 24

5 Implementation and Results 25
5.1 Choice of Simulation Actors 25
5.2 Choice of Oracle 25
5.3 Implementation 27
5.3.1 The Market Actors 27
5.3.2 The Oracle 29
5.3.3 Coverage 30
5.4 Results and Analysis 31
5.4.1 Simulations 31
5.4.2 MTBF of the System Under Test 32
5.4.3 Bugs Discovered in the Common Data Server 33
5.4.4 The TAX Bug 33
5.4.5 MTTF for the TAX-crashing Bug 33
5.4.6 Oracle Analysis 36

6 Discussion 37
6.1 Conclusion 38
6.2 Further Work 38

Chapter 1

Introduction

1.1 Background

A trading system is very complex and difficult to test. Manual and automatic tests can be used to find errors but far from all bugs will be found by these conventional tests. By using random tests, through trading simulations with randomly generated data, low-frequency errors that would cause the system to fail can be detected. A crash that causes trading to stop, if even just for a couple of hours, can be extremely costly, making reliability very important.

Today many stock exchanges around the world are run privately and are subject to competition. Trading systems must meet the demands of brokers, private traders and other customers at the stock exchange. In order to meet these demands, the trading system must be reliable. This can only be achieved through rigorous testing of the system.

1.2 Problem Description

The goal of this thesis is to conduct a theoretical study of random tests and to develop a random test using trading simulations in order to find errors in a trading system. The purpose of this is to introduce random tests into the testing process at Cinnober Financial Technology.

In this thesis I wish to test the hypothesis that random tests can find low-frequency bugs that have not been found by other automatic and manual tests. This will be done by performing random tests on an already tested trading system. The tests will consist of trading simulations, using actors such as traders and market operators. Simulations will be run allowing actors to perform various transactions in the trading system. The test will be analyzed using a test oracle. Different types of oracles will be studied in order to choose a suitable oracle for random testing a

market place system.

1.3 Overview

This thesis is divided into two parts; a theoretical study of random testing and a practical study consisting of the development of a random test in a trading system. The theoretical part is covered in chapters two, three and four. The practical study is discussed in chapters five and six.

The first chapter gives a short introduction to the subject as well as a description of the assignment.

Chapter Two gives a brief introduction to markets and stock exchanges as well as a short description of TRADExpress, the trading engine developed by Cinnober Financial Technology.

Chapter Three is an introduction to software testing in general.

The fourth chapter explains random testing. Test oracles as well as trading simulations are discussed in this chapter.

Chapter Five contains a description of the implementation and the results that were obtained as well as an analysis of these results.

Chapter Six gives a discussion of the project and suggestions for further work.

1.4 Acknowledgments

I would like tho thank my supervisor at Cinnober, Lars Wahlberg, for the time and effort he has put into guiding me through my project and teaching me about software testing. He has been incredibly helpful and his guidance has been invaluable.

Chapter 2

The Market Place System

A market is the means through which buyers and sellers are brought together to transfer goods and services. The stock exchange describes the place where these participants can trade goods and securities.[Schauer 2006]

2.1 Market Place Actors

There are several types of participants or actors at an exchange, each with specific roles. These can be divided into three major categories: Customers, brokers and market makers.[Schauer 2006]

2.1.1 Customers

Customers are people and institutions who trade shares in order to invest their money. The biggest difference between retail customers and institutional customers is that retail customers generally trade smaller amounts.[Schauer 2006]

2.1.2 Brokers

A broker is an agent who facilitates trades for their customers. Brokers do not take any risks in a trade since they simply act as middlemen between the customer and the exchange. Brokers can sometimes also provide other services such as investment advice and portfolio management.[Schauer 2006]

2.1.3 Market Makers and Specialists

Market makers maintain a firm bid and ask price in a security by offering to sell or buy at quoted prices. Market makers add liquidity to the market by always being ready to buy and sell. Market makers make their money by buying at the lower bid price and selling at the higher ask price.[Schauer 2006]

2.2 Instruments

A financial instrument or security is traded on a market. Examples of instruments are stocks, bonds, indexes and commodities. These are often referred to as underlying instruments. Another type of instrument is derivatives. Examples of derivatives are options, forwards, and futures. Derivatives can be based on underlying securities or other derivatives.

2.3 Orders and Order Books

2.3.1 Orders

An order at a stock exchange is an instruction from a customer to buy or sell shares. There are many different types of orders. Each order has it's own characteristics and not all order types are allowed at every stock exchange. The orders available depend on the rules of the specific exchange.[Schauer 2006] Some of the different order types are mentioned below.

Market order

A market order is a buy or sell order without a price limit. The customer states the number of shares to buy or sell but not the price. A market order will immediately be executed at the best available price. A market order is always executed at the best price no matter how bad that price may be for the investor. Market orders have the advantage that they have the highest priority and therefore are matched first.[Schauer 2006]

Limit Order

A limit order is a market order with a price limit. This is an instruction to buy or sell shares at a specified price or better. Limit orders are more often used to buy securities than to sell them.[Schauer 2006]

Stop Loss Order

A stop-loss order is an order with a specific trigger price at which the order converts to a market order. There are also stop-limit orders that turn into limit orders when the trigger price is reached or exceeded.[Schauer 2006]

Time Limit Orders

Limit orders can be valid until a specified time or event. There are several types of time limit orders. A "Good for day order" is valid the rest of the trading day. a "Good till cancel" order is valid until it is executed or canceled by the customer. The validity periods are listed in table 2.1.

Validity Period	Description
Good till cancel	The order is valid until canceled.
Good for day	The order is valid for the rest of the trading day.
Good till date	The order is valid until a specified date or time.
Good till next uncross	The order is valid until the next uncross.
Fill and kill	The Fill and Kill order is described below.
Fill or kill	The Fill or Kill order is described below.

Table 2.1. Validity Periods for Limit Orders

Combination Orders

A combination order is an order to buy/sell an amount of one security and to buy/sell another amount of another security. Combination orders make it possible to use complex trading strategies.[Gundemark 2005]

Linked Orders

A linked order is an order to buy/sell an amount of one security or to buy/sell another amount of another security.

Fill-Or-Kill

A FOK-order will only be executed if it can be executed fully. Otherwise the order will be deleted.

Fill-And-Kill

A FAK-order is like the FOK-order except that it can be filled partially. The rest of the order is then canceled. this is also called an Immediate or cancel order.

Limit-or-better

A limit-or-better order is only necessary if the market is better than the quoted limit.

All-Or-None

An AON-order is similar to a FOK-order, the only difference being that an AON-order is not canceled if it is not executed immediately.

Hidden Volume Orders

A hidden volume order is an order where only part of the order is visible.

2.3.2 Order Matching

Order matching is when an order is executed with another order, resulting in a transaction. This matching mechanism is strongly influenced by the trading system (quote driven or order driven market). Orders are matched according to their price and priority.[Schauer 2006]

2.3.3 Order books

Order books play a central role in any exchange. An order book lists all different types of orders for a certain security. An order book contains all buy and sell orders and other order types. This is important for order matching since some order types have higher priority than others. All new orders during the trading session are examined automatically, even those that cannot be executed right away. Orders that can't be executed remain in the order book until they can be executed.[Schauer 2006]

2.3.4 Transparency

Order transparency indicates how much information about an order that is public. The highest level of order transparency is Market-By-Order where every order, including price and volume is shown in the order book. Market-By-Level on the other hand aggregates all orders with one price and shows the total volume for each level. Counter party info transparency can be set to show no or full information about a counter party.[Gundemark 2005]

2.4 The Trading System

Cinnober Financial Technology's trade platform TRADExpress provides functionality for a market place system. This section describes some of this functionality in order to give an overview of how the system works.

2.4.1 Trading States

The trading schedule consists of a sequence of trading states. An example of a simplified 24 hour schedule taken from [TE 2007] can be seen in table 2.2.
Several attributes are configured for each trading state, such as whether automatic matching occurs or not and what transactions are allowed in the state.[TE 2007]

2.4.2 Order Book Rule Groups

A group of order books that follow the same trade schedule often have the same parameters. To simplify management Order book rule groups are defined. Market operations can set parameters on the group level. Attributes tied to Order book rule

Time	Trading State
0h	System Closed
+7h	Pre-Trade
+8h	Continuous trading
+15h	Post Trade
+16h	System Closed
+24h	End of Cycle

Table 2.2. An example of a Trading Schedule

groups are tick size table, reversed prices or not, trading schedule, Market-by-level or Market-by-order, and anonymous or public.[TE 2007]

2.4.3 Orders

The order management component of TRADExpress captures all incoming order actions and responds to them. The available order actions are listed in the table below (taken from *TRADExpress Trading Engine Overview* 2007)

Order Action	Description
Order Insert	Insert a new order.
Update Order	Modify an existing order.
Cancel Order	Cancels an existing order.
Cancel All Orders	Cancels all orders for either the user, participant or order book in question
Suspend Order	Withdraws orders from the order book but lets them remain in the matching engine
Activate Order	Activate suspended orders.

Table 2.3. Available order actions in the trading system

The allowed order types are limit orders, time limit orders such as ”Good for Day” and ”Good till Cancel” orders, Fill and Kill orders, Fill or Kill orders, stop loss orders, and hidden volume orders. The trading engine also has functionality for a variation of combination orders.

2.4.4 Execution of Transactions

When the system receives orders, they can be automatically matched. Automatic matching allows incoming orders to be matched with orders already in the book. If the order is not completely filled then it is booked into the order book, unless the order is specified to cancel the remainder. The execution price is determined by the order already in the book.[TE 2007] An order book can be set up to run without automatic matching, specified to execute with another specific order. When

opening, re-opening or closing a market an uncrossing auction is used. If there are crossing prices in an order book, an auction price is calculated.

2.4.5 Market Management

Authorized personnel may manage different parts of the trading engine. They may for instance add order books, halt trading for a specified order book, lift trade halts and maintain the trading schedule. Participants can be added, edited, enabled and disabled. Order book rule groups can be added, changed and removed.

2.4.6 The System's Components

The Systems's components are shown in figure 2.2. The trading system created by Cinnober Financial Technology, TRADExpress has the following components:

Matching Engine (ME)

The matching engine maintains order books and active orders. The ME matches orders to generate trades. The ME is multi threaded allowing several transactions to be worked on at the same time. [TE 2007]

Daemon

There is a daemon in every box that runs TRADExpress and it handles all the servers in that box. [TE 2007]

Vote Server (VS)

The Vote Server decides which server is primary and which server is standby. If the primary and standby servers lose their connection to each other, the vote server will make sure that the standby server does not become primary. [TE 2007] The VS is connected to the servers according to figure 2.1.

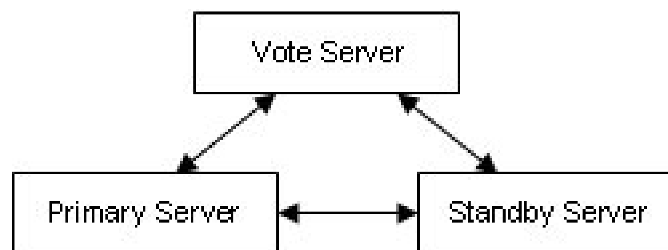

Figure 2.1. The Vote Server

Trading Application Multiplexer (TAX)

The TAX acts as a message router for both incoming and outgoing traffic. Market participants connect to a TAX gateway. [TE 2007]

Common Data Server(CD)

The Common Data Server(CD) contains a user database, product definitions, authorization information, trading schedules etc. The CD uses a database. [TE 2007]

Query Server (QS)

The Query server maintains a copy of the active order books and orders in order to offload the ME from large queries. [TE 2007]

History Server (HS)

The History Server persists overnight orders and trades and serves queries for historical information. [TE 2007]

Management Repository (MR)

The MR maintains system operations data such a s status events and statistics. [TE 2007]

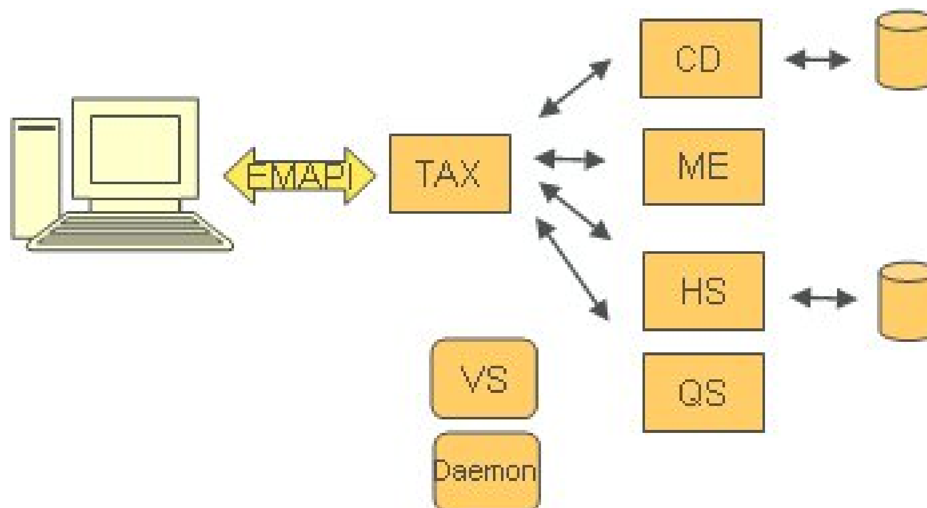

Figure 2.2. overview of the trading system

Chapter 3

Software Testing

Testing software improves the quality of the software by identifying and removing defects through debugging. The removal of identified bugs is called debugging and is not the same thing as testing. While debugging is concerned with localizing and correcting faults, the goal of testing is the detection of failures that indicate the presence of bugs. [Spillner 2006]

Testing can show that a program contains defects but cannot prove that a program is defect free. In order to do this, the program would need to be executed in every possible way, with every possible data value, which of course is impossible for almost any program. Testing reduces the risk that defects are hidden in the program, but even if no failures are found, this is no proof that defects are not present in the program. Since exhaustive tests are impossible, the test effort must be controlled taking into account risks and priorities. [Spillner 2006]

When testing a software system, several properties may need to be verified. These properties are often categorized as functional and non-functional properties. A functional test verifies the output of a system while a non-functional test might test that external requirements are met such as timing and resource usage. [Zhan 2002] In this thesis the main focus will be on functional properties. figure 3.1 shows a diagram of the different test methods described in this chapter. Statistical tests are described in chapter four.

3.1 Static Testing

One test method is the static test, consisting of manual checking and static analysis. In a static test, the test object is not executed with test data but instead analyzed. This analysis can consist of inspecting all documents in a software project manually. The main goal of the examination is to find defects and deviations from existing specifications. The basic idea of static analysis is to find defects and deviations as early as possible in the development process. [Spillner 2006]

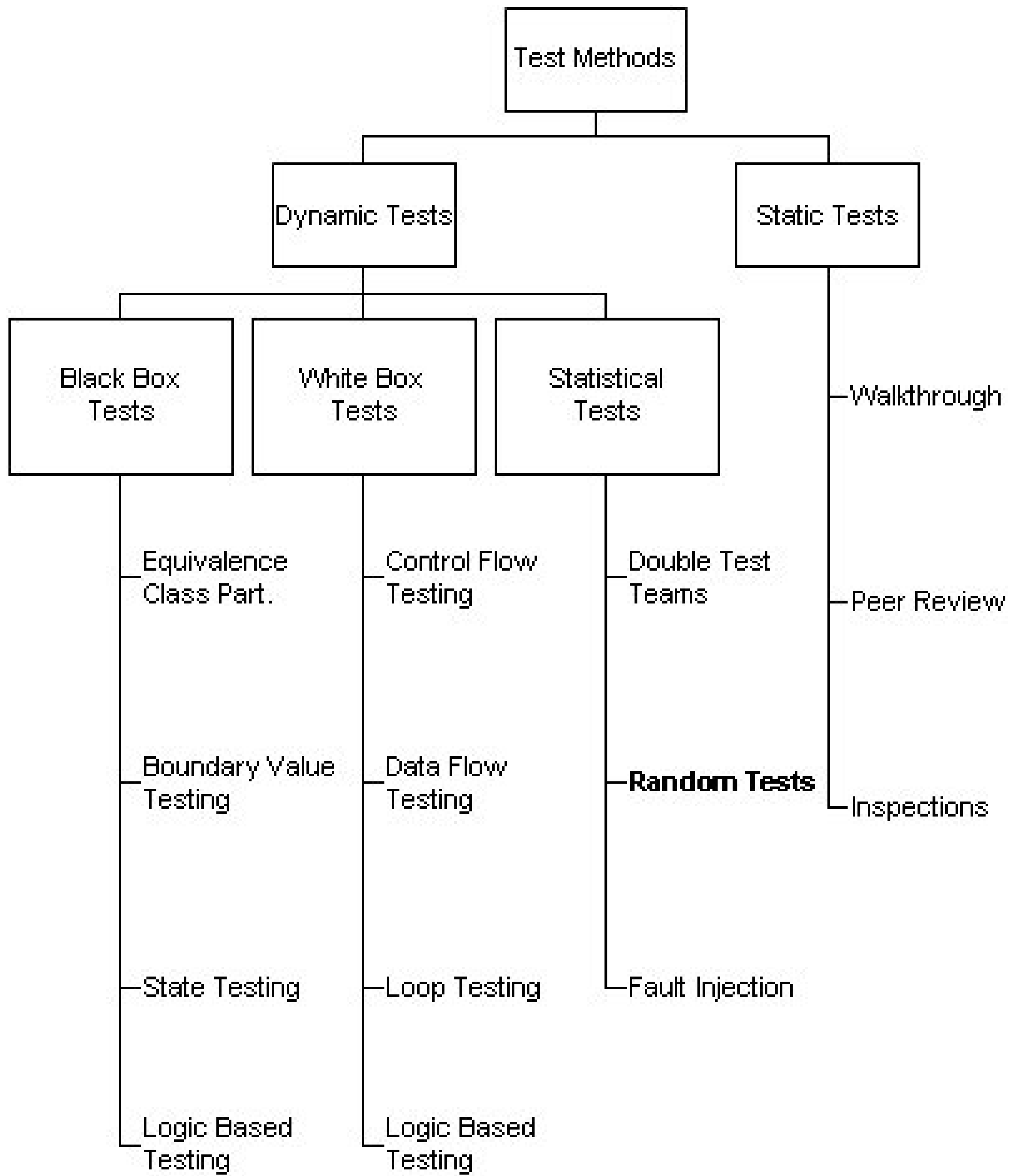

Figure 3.1. Different types of tests

Walkthrough

One static method is the walkthrough. A walkthrough is an informal review method which is used to find problems in the written documentation. The author presents the documents to reviewers in a meeting. The author presents the product and typical use cases are walked through according to the course of events. [Spillner 2006]

Peer Review

In a peer review, the colleagues of the author give feedback on the document subjected to the review. Peer reviews are an efficient means to assure the quality of the examined documents. [Spillner 2006]

Inspections

The inspection is a more formal review. Every person involved in an inspection has a certain defined role. The inspectors prepare themselves using procedures, standards and checklists. An inspection meeting is held and lead by a moderator. The moderator makes sure that everyone is prepared for the meeting and then issues brought up are discussed. In an inspection, data is also collected in order to assess the quality of the development and inspection processes. [Spillner 2006]

3.2 Dynamic Testing

Dynamic tests are an execution of the test object on a computer. The system under test must be executable. It is provided with input data and different parts of the program are called on through a predefined interface. One must determine the conditions and preconditions for the test as well as the goals to be achieved. Each individual test case must be specified and it must be determined how to execute the tests. There are several different approaches to dynamic testing. They are usually divided into black box and white box testing. [Spillner 2006]

3.2.1 Black Box Testing

Black box testing is based on specifications and its main objective is to ensure that every requirement is actually fulfilled [Fu 2003]. Black box tests are done without knowledge or consideration of the inner structure and design of the system under test. Black box testing is also known as functional or behavioral testing, because of the observation of input and output behavior.[Spillner 2006] The objective of black box testing is to find out when the input-output behavior of the program does not agree with its specification. One strength of black box testing is that tests can be derived early in the development cycle. The system under test is treated as a black box and its functionality is tested by feeding it with various combinations of input data.[Sthamer 1995]

Equivalence Class Partitioning

An equivalence class is a group of data values that are assumed to be processed in the same way by the system under test. It is considered to be sufficient to test one representative of each equivalence class since it is assumed that the reaction will be the same for any input value in that equivalence class. Besides equivalence classes for correct input, those for incorrect input must be tested as well. [Spillner 2006]

Boundary Value Testing

Boundary value analysis is testing with input data close to the boundaries of different equivalence classes. Faults often appear around these boundaries since they are not always clearly defined and programmers sometimes misunderstand them. On every border, the exact boundary value and adjacent values are tested. Failures are often found with these tests. [Spillner 2006] In the 2-dimensional case boundaries are tested at the boundaries of both variables as well as at the boundaries of each variable. The same method can be applied to the n-dimensional case. Non-numeric data can also be tested using this method. If the input for example is taken from an array of strings, the first and last string can be seen as boundary values.

State Transition Testing

In a state transition test, the test object starts from an initial state and can then come into different states. when performing state transition tests all possible states should be reached and the test should execute all specified functions of a state at least once. Invalid state transitions also need to be tested. [Spillner 2006]

3.2.2 White Box Testing

In White Box Tests or structural tests, the designer considers the structure of the system under test. Component hierarchy, flow control, data flow etc. is considered.[Spillner 2006] The structure of the software is examined by execution of the code and test data are derived from the program's logic.[Sthamer 1995]

3.2.3 White Box Analysis

White Box Analysis can be used to measure test coverage. Test coverage measures the amount of testing performed by a set of tests.[Graham 2007] There are several ways to measure coverage. Theses methods are briefly described below.

Code Coverage

Code coverage is an analysis method that determines which parts of the software have been executed by the test. Code coverage includes statement coverage, decision coverage and condition coverage. [Graham 2007]

Statement Coverage

Statement coverage is the portion of executable statements that have been exercised by a test. [Graham 2007]

Decision Coverage

Decision coverage is the percentage of decision outcomes that have been exercised by a test. [Graham 2007]

3.3 Experience-Based Testing

Although testing should be rigorous, thorough and systematic, there is a role for non-systematic techniques based on a persons knowledge, experience and imagination. Some defects are difficult to find using more systematic approaches.

3.3.1 Exploratory Testing

Exploratory testing is test design and test execution at the same time. This is the opposite of scripted testing where tests are carried out according to a predefined test procedure [Bach 2001]. Any testing where the tester continuously designs the test as the test is being performed, using information gained while testing, is an exploratory test [Bach 2003]. Exploratory testing can be applied to testing a market place system by allowing several testers to manually simulate actors on the market, each tester given a few general tasks to perform.

3.3.2 Error Guessing

Error guessing is a test design technique where the experience of the tester is used to anticipate what defects could exist in the system under test.[Graham 2007] An experienced tester with good knowledge of the system and the tests that have been performed can have a good idea of what parts of the system are likely to contain errors.

3.4 Automatic Testing

Doing repetitive work manually is tedious and boring which causes people to make mistakes when doing the same task over and over. Examples of this kind of repetitive work include running regression test and entering the same data over and over again. Test tasks can instead be performed by automated test tools. Some benefits of using automatic test tools are: [Graham 2007]

- Reduction of repetitive work.
- Greater consistency and repeatability.

- Objective assessment.
- Ease of access to information about tests or testing.

One type of test that is performed in a market place system is the trivial test. A trivial test is a very simple test that checks a . A trivial test can for example check conditions such as that mandatory fields and data types are fulfilled. For such test cases automated tests are used since performing such tests manually would be time consuming and boring, with an increased risk of typos when specifying test cases. Another reason to use automated tests is that in order to get sufficient test coverage of a complex system such as a trading system, a very large amount of test cases are needed.[Sellberg 2003]

The amount of automation used in a test can vary from completely manual tests with no automation at all to tests that run and self-verify completely automatically. In between the manual tester can get many automated assists. What level of automation to use depends on the specific situation.[Hoffman 2000]

Chapter 4

Random Tests

4.1 Introduction - Concept of Random Tests

The general idea of random testing (also known as statistical or Monte-Carlo testing) is to send randomly generated data to the system under test. The input data is generated from a predefined domain. The output can then be analyzed by an oracle which decides whether the output is correct or incorrect.[Gundemark 2005] Different types of oracle are discussed later in this chapter.

Automatic tests that do the same thing every time they are run are most likely to find a software defect the first time they are run.[Hoffman 2000] Running the same automated test over and over again is not likely to uncover new defects unless the system under test is changed, which is the purpose of many automatic tests. By introducing randomness to a test we increase the probability of finding new errors even after running the test many times.

Some of the same techniques used for regular automated tests can also be used in random tests. Some of the most common of these practices are partitioning, boundary value analysis, probabilistic testing and error seeding.[Gundemark 2005] Partitioning is when the input and output are divided into smaller partitions, and tests are performed on each partition to verify correct functionality. Both valid and invalid input data can be tested this way. Boundary value analysis is designing test cases for data close to the boundaries of a partition. Faults often appear around these boundaries since they are not always clearly defined and programmers sometimes misunderstand them. [Spillner 2006]. Probabilistic testing is designed to test the system's stability rather than to find bugs, and is important for safety critical systems, which have to be extremely reliable. Error seeding is done by introducing errors in the code. By running tests and measuring the amount of found injected faults an approximation of the total amount of errors is obtained.

By combining random tests, one can create tests that are increasingly complicated,

which is useful as the system under test increases in stability. By using more extensive oracles, planned regression testing can be replaced by a combination of manual exploratory tests and automated random tests.[Kaner 2000]

4.2 Data Generation / Input

4.2.1 Distribution

[Gundemark 2005] When randomly generating input data for a test it might be a good idea to use several distributions, generating both realistic and unrealistic data. Using realistic data will measure the stability of the system and using unrealistic data will increase the probability of finding low-frequency bugs.[Gundemark 2005]

One way of obtaining realistic data is to simply use historical data such as earlier trading days in a market place system. This will allow thorough tests of existing functions. New functionality can only be tested if data is generated using an alternate method. By using realistic data one can measure the stability of the system. One such measurement that is frequently used is Mean Time To Failure(MTTF). MTTF is the measure of expected time from a failure free state to a failure.

Generating unrealistic random data and using it as input to the system under test, may produce unusual results such as a series of transactions that very rarely happen on an ordinary trading day. This allows the test to find low-frequency errors described in section 4.5.2, that might have passed a function test.

4.2.2 Domain

The domain from which random input data is generated is an important aspect of random tests. The input domain for a complex system is practically infinite, which is why one might use data from a select sub domain. This makes it possible to do extensive testing with a limited number of test cases. Different parts of the system's functionality will be tested when using data from different sub domains. It is also possible to send incorrect data to the system in order to test error handling.[Gundemark 2005]

4.3 Simulations

4.3.1 Trading Simulations

Trading simulations allow us to create very complex trading scenarios in order to increase test coverage. Together with trivial tests, trading simulations can be a powerful tool.[Sellberg 2003]

A transaction in a market place system can have a very large number of results,

depending on the contents and configuration of relevant order books.[Sellberg 2003] Market place systems can support the use of special order types such as stop-loss orders and combination orders with which just one order can initiate a matching sequence including hundreds of different orders in different order books.[Gundemark 2005]

One major difficulty in using trading simulations is evaluating them. The interaction between the actors is very complex, making the simulations hard to evaluate. There are however three types of evaluations that can be performed:

- Trivial tests.
- Certain types of explicit tests can be applied to the actors.
- The system under test does not crash.

4.3.2 Modeling Actors/Human traders

Market participants can be modeled as *actors*, where each actor has a specific usage profile. The profile consists of a number of actions, each given a probability. Parameters such as order size and distance to spread are randomized. Actors typically have 10-30 different actions each with two or three randomized parameters. Simulations can be run with several different actors, all with different actions and probabilities. The actors will then act like market participants, acting independently basing their actions only on their own profiles and the current market condition. This allows the actors to trade with each other.[Sellberg 2003] Table 4.1 shows examples of trading simulation actors taken from Gundemark [2005]. The actors can be randomized

Actor	**Description**
Trading Control	Suspend/Release order books, flush different order types, stop order books, delete orders, suspend participants etc.
State Administrator	Changes the state of order books
Sell Trader Buy Trader	Different Sell/Ask orders at different states, update/suspend/activate orders, make queries etc.
Hidden volume Trader	Enters different Hidden volume orders, absolute or relative price updates, order actions, update/suspend/activate orders etc.
Stop-Loss trader	Enters different Stop-loss orders, absolute or relative price updates, order actions, update/suspend/activate orders etc.
Actor Combination Actor Linked	Enters Combination or Linked orders, update/suspend/activate orders etc.

Table 4.1. Examples of trading Simulation Actors

using a seed, allowing the tester to run the same simulation several times. Another method of modeling human traders is through manual exploratory tests with several testers manually acting as traders.

4.4 Oracles

In order to verify test results an oracle can be used. The purpose of an oracle is to capture and compare actual with expected results. An oracle can be expected results, the process of generating expected results or a program, or mechanism used to generate expected results [Hoffman 1999]. Actual test results are then compared to the oracle's results to verify their correctness. Some interesting characteristics that relate the oracle to the system under test are listed by Hoffman [2001]:

- Completeness of Information
- Accuracy of Information
- Usability of the oracle
- Maintainability of the oracle
- Complexity
- Temporal relationships
- Costs

Completeness of information consists of input coverage, result coverage, function coverage, sufficiency and types of errors possible. Completeness can range from no predictions to an exact copy of the system under test. With completeness and accuracy comes complexity which can cause the oracle to become slow and expensive. Also a complex oracle is more likely to contain errors and is harder to maintain. An oracle that predicts more about program state and environment conditions is also more sensitive to changes in the system under test and operating environment. Errors can be missed because of the oracle sharing a component with the system where both the oracle and the system under test may return the same incorrect output.[Hoffman 2001] Conditions such as test environment and program state must be considered before comparing test results to the oracle.

There are a very wide variety of test oracles. Some of these which are discussed in [Hoffman 2001] are: no oracle, true oracle, consistent oracle, self-Referential oracle and heuristic oracle.

4.4.1 No Oracle

Running automated tests without checking the results has the disadvantage of only discovering spectacular errors that cause the system to crash. On the other hand they are inexpensive, easy to implement and run fast.[Hoffman 2001] Using a "No oracle" strategy is effective only when searching for bugs that cause the system to crash.

4.4.2 True Oracle

A True oracle reproduces the same results as the system under test. The same data is sent to the oracle as to the system under test, and the results are verified using separate algorithms, platform, processes etc. For any given test case all values are verified by the oracle's separate algorithm. The less the oracle has in common with the system under test, the less likely it is that errors will be shared by the system under test and the oracle. Such errors will probably go unnoticed and an oracle with less in common with the system under test will have a higher level of confidence in the correctness of the results. True oracles are usually slow and expensive to use. Tests using true oracles may use a small sample with a limited amount of test data. One way of doing this is through Random Tests. By using a ”random” seed, the same data can be sent to the oracle for verification.[Hoffman 2001]

4.4.3 Consistent Oracle

A Consistent oracle uses test results from a previous test as an oracle. The Consistent oracle can be an older version of the system under test, an equivalent program or software from another platform. One of the major advantages with the Consistent oracle is that it is the fastest method using an oracle. Consistent oracles are often used for regression tests and are suitable for testing changes from one revision to another. Since we don't need to know whether the results are correct we can generate huge amounts of data to be tested. Consistent oracles will find any changes and work well to discover new problems but will not find old problems that were there before changes were made.[Hoffman 2001] A problem that can arise when using such an oracle in a market place system is that the same input can give a different output in different versions of the system.

4.4.4 Self-Referential Oracle

When using a Self-Referential oracle, the results are built in to the input data. For example, in a test for a data base engine, a data field could describe the data base linkages expected between fields or records. A random number seed can be included in the data so the test can be rerun with the same random number series. When using a Self-Referential strategy, tests create records with specific characteristics, which are included within the records themselves. The advantages of a Self-Referential oracle are that one can generate and verify large amounts of complex data and that it allows extensive post-test analysis.[Hoffman 2001]

4.4.5 Heuristic Oracle

A Heuristic oracle uses a heuristic to quickly assess whether the result is likely to be correct or not. Instead of checking exact results a heuristic oracle verifies results using simpler algorithms based on a heuristic. For example if a field should contain a Swedish postal code, a heuristic strategy might be to check that it has five digits. A

heuristic for the function $sin(x)$ could be to check the identity $sin^2(x)+cos^2(x) = 1$ or to check that it is strictly increasing in the interval $(0, \frac{\pi}{2})$ and decreasing in the interval $(\frac{\pi}{2}, \frac{3\pi}{2})$ which can be checked by comparing a few points within these intervals. One might also check the exact results for a few select points such as the points where $sin(x)$ is equal to zero or one. Such a heuristic oracle is much easier to implement than an alternate version of the sine function and runs much faster.[Hoffman 1999]

One trick to creating a heuristic oracle is to divide the system under test into several smaller pieces.[Hoffman 1999] For example in the $sin(x)$ example the function is broken into two ranges for the heuristic: rising and falling results. Another trick is to find other simple relationships between input variables and results related to those being tested.

Most complex algorithms contain simple patterns. By looking at the relationship between input and results, one can find approximations or observations that give a suitable heuristic. The simpler the algorithm is, the faster and more reliable the heuristic oracle. In some cases for example in GUIs, there is no simple pattern to work from and heuristic oracles won't apply. If one uses overly complex patterns, new errors can be introduced causing false error reports.[Hoffman 1999]

4.5 Reliability

Measuring Software reliability is difficult. Those working with reliability are not in agreement on how to measure the reliability of software. Some engineers argue that statistical methods cannot be used since faults will occur every time a certain part of the code is executed, the faults are deterministic and cannot be modeled using probability. Others claim that in a complex system a fault can take an infinite number of forms, meaning that undetected errors can be seen as random. So far, those opposing statistical analysis have not come up with an alternative method.[Gundemark 2005]

4.5.1 Models for Software Reliability

There are several ways to measure the reliability of a system. Software reliability is often defined as the probability of failure free software operation for a specified period of time.[NASA 2004]The primary goal of reliability modeling is to calculate the probability that the system will fail in a given time interval, or, the expected duration between successive failures. software reliability models can be divided into prediction models and estimation models. Prediction models attempt to predict how reliable a system will be when it is completed. Estimation models attempt to estimate how many defects still remain in the system or the time between failures. Estimation models include reliability growth models, input domain models and fault seeding models.[NASA 2004]

Another method is to measure the failure rate λ, of a system. The failure rate is defined as the number of expected failures per hour. The failure rate can be seen as time-invariant(failure rate is constant over time) to simplify calculations allowing us to model the failure rate as Poisson distributed. When viewing the failure rate as time-variant, a Weibull distribution can be used.[Gundemark 2005]

One may argue that software failure cannot be random since a specific set of inputs will always cause the same failure. However if the input to the system is random, software failure can be modeled as a random variable.[Johansson 1996] If we measure the time between each failure we can estimate the Mean Time Between Failure(MTBF). This gives us an estimate of the failure rate λ, where of course $\lambda = \frac{1}{MTBF}$. This allows us to model the failure rate as a Poisson approximation where the number of failures over a period of time can be viewed as a random variable $X \in Po(\lambda)$, [Wååk 1978] meaning that the probability of k faults occurring in the time T is

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

The time between failures can then be modeled as a an exponentially distributed random variable $Y \in Exp(\lambda)$. The mean of this random variable is equal to the MTBF.

One can use so called fault count modules in the debugging phase to model the number of faults found in given test intervals. Since software does not wear out over time, one can assume that as errors are removed, the reliability increases as long as new errors are not entered into the system. Most models are based on a Poisson distribution whose parameter takes different forms for different models. [Johansson 1996]

The fault seeding model introduces faults into the system. By counting the number of seeded faults found an estimation of the total amount of faults can be obtained.

Markov models use the modular nature of most high-level programming languages to calculate the reliability of a program from the reliability of individual modules to other modules.[Johansson 1996]

Another model is to estimate the number of bugs per line of code. This model assumes that for each specific language, the number of bugs per line of code is constant. Such models are not considered very reliable but can be used to give a first estimate of reliability. [Johansson 1996]

According to Johansson [1996], there are a few assumptions used in software reliability modeling. The first assumption is that times between failures are independent. This assumption is used in all time between failure models, but is not always valid

if the tester is concentrating tests to a failure-prone section of the code.

Another assumption is that no new faults are introduced during the fault removal process. Removing one fault may lead to another fault being entered in to the system.

One often assumes that failure rate is proportional to the number of remaining faults. This implies that each remaining fault has the same probability of being detected, which is not always the case.

Another assumption is that reliability is a function of the number of remaining faults which also is untrue since some portions of the code are more likely to be executed than others.
The final assumption mentioned by Johansson, is that the failure rate increases between failures. This would be a justifiable assumption if software wore out over time, which it does not.

4.5.2 Low-Frequency Errors

As mentioned earlier, one purpose of random tests is to find low-frequency bugs that go undetected by other tests. In this section, an attempt will be made to answer the question: What is a low-frequency error or bug?

A low-frequency error is an error that does not occur often. The low-frequency bugs discussed in this thesis, although rare, occur often enough that it would be cost-efficient to fix them. Table 4.2, taken from [MIL 2000], describes probability levels of mishaps.

Desription	**Probability of occurance in the life time of the system**
Frequent	$> 10^{-1}$
Probable	$10^{-2} - 10^{-1}$
Occasional	$10^{-3} - 10^{-2}$
Remote	$10^{-6} - 10^{-3}$
Improbable	$< 10^{-6}$

Table 4.2. Probability levels for errors

For the purpose of this thesis, low-frequency errors are defined as errors that occur with a probability of less than 10^{-2} in the lifetime of a trading system, that is errors that according to MIL [2000] are occasional, remote or improbable. A trading system can be assumed to have a lifetime of approximately five years.

Chapter 5

Implementation and Results

The purpose of this project is to test a trading system using random tests. The tests are done through simulations of actors, or market participants. Various orders are sent in to the system and the result is verified by an oracle.

5.1 Choice of Simulation Actors

The simulation consists of several actors with varying attributes. Actors can be different types of traders or market managers. A simulation can be implemented using one thread for each actor allowing many actors to simultaneously enter a large number of transactions, or by using only one thread in order to repeat exact trade sequences where something went wrong. The first method allows a very large number of simulations in a short period of time while the advantage of the second method is that it is easier to locate functional bugs when they are discovered. By first using only one thread until most of the bugs have been found and then implementing a more realistic simulation using many threads, many bugs may be found. The combination of the two methods mentioned above may prove to be a powerful tool. This project mainly focuses on multi-threaded simulations with many actors acting simultaneously.

5.2 Choice of Oracle

As I described in the previous chapter, there are several types of oracles that can be used to verify tests. Choosing an oracle is not easy and I will go through the five types of oracles I mentioned in Chapter 4, focusing on how they can be used in a trading system:

No oracle

Using a No oracle in such a complex system as a trading system has it's advantages. Random tests often find low-frequency bugs that can crash the system. Since these

often are the most important errors to find, using a No oracle strategy can be sufficient. This is the least expensive oracle, it contains no errors in itself and finds bugs that cause the system to crash. However, this strategy will not find more subtle errors that don't crash the system.

True oracle

It is possible to create a trading system independently without knowledge of the architecture of the system. This can be done by developers that have no knowledge of how the system was developed or designed. They can then build a completely separate system with only the requirements in common with the original system. Data can then be sent into both of the systems in order to compare the results. This will find many bugs but designing such an oracle is expensive, and time-consuming. There is a risk of errors existing in the oracle, and the oracle in itself might need extensive testing. However, errors in a trading system are expensive and spending money on a good oracle may pay off.

Consistent oracle

Using a consistent oracle in a trading system can be difficult since the system is multi-threaded until the orders enter the matching engine. This means that even if the same transactions are sent in to the system, they will not always give the same results.

Self-referential oracle

The main difficulty of using a self-referential oracle strategy in trading system is that one must be able to send expected results with the data. Such results do not always exist in a trading system. for example if we send an order insert transaction, we do not know if or how the order will be traded.

Heuristic oracle

When performing random tests, random data is sent into the system making it difficult to predict exact results. However we can have a general idea of what the results can be, making a heuristic oracle a good choice for a trading system. A heuristic oracle is less expensive than a true oracle and can find more errors than a no oracle strategy.

The Oracle Used in this Project

The oracle chosen to be designed most resembles a heuristic oracle since this project is mainly concerned with checking that certain requirements are met. These requirements can be boundary value tests such as making sure that a negative price has not been entered in an order or that the size of an order is greater than zero. The

oracle will also be able to check that illegal operations, such as that trading in a trade halted order book, have not been performed. A detailed description of what checks the oracle performs can be found in the following section. The oracle should also be able to communicate with the actors, the actors will send orders both to the oracle and the system, so the oracle can check that orders are matched and traded as expected.

5.3 Implementation

The tests were implemented as simulations of trading in a market place system. The simulations consist of market actors such as traders and market operators acting on the market by trading and performing market actions. An oracle listens to the actors as well as to the system in order to determine if something has gone wrong. Also system failures such as crashes work as a sort of oracle. Simulations are run by letting the actors perform their actions randomly. Many actions are performed in each simulation. An overview of the test is shown in figure 5.1.

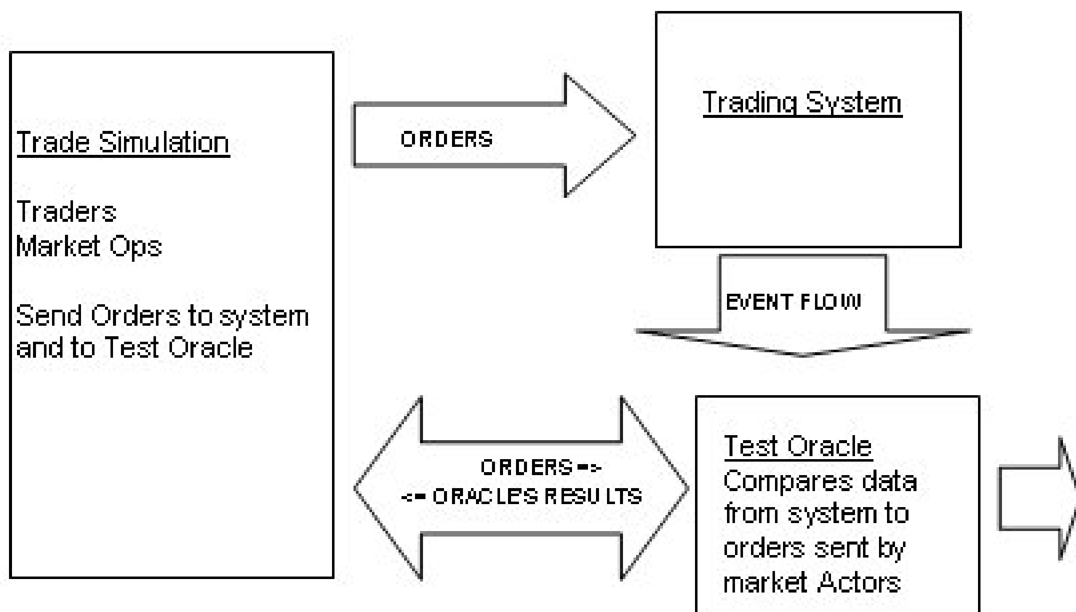

Figure 5.1. A Trading simulation using market actors and an oracle.

5.3.1 The Market Actors

Several Actors are used in the simulation. Actors have different roles such as regular traders, Stop-Loss traders, hidden volume traders, market operators and market makers. Each market actor has a set of actions, performing each action at a given probability. Each actor performs a given amount of actions before logging out of

the system. When an actor performs an action it may also send information to the oracle. The following actors are used in the simulation:

Buy Trader

The *Buy Trader* inserts bid orders close to the last traded price. The bid price is approximately $N(P - 0.1, 0.5)$ distributed where P is the last traded price. The volume of the order is an approximately uniformly distributed integer between 1 and 100. The *Buy Trader* can also cancel it's orders. With a given probability each inserted order is later canceled. If the order has been matched, the actor will still try to cancel it, expecting an error message. The *Buy Trader* can also be set to Mass Cancel it's orders, canceling all of the actor's orders given that they haven't been matched. The validity period is pseudo-randomly chosen and can result in the order being a Fill-And-Kill order, Fill-Or-Kill order, Good to Cancel or any of the other validity periods described in chapter two.

Sell Trader

The *Sell Trader* is very similar to the *Buy Trader* but instead enters ask orders with an ask price that is approximately $N(P + 0.1, 0.5)$ distributed where P is the last traded price. The *Sell Trader* cancels and mass cancels orders in the same way as the *Buy Trader*.

Hidden Volume Trader

The *Hidden Volume Trader* acts as a regular buy or sell trader except that it's orders are so called *Iceberg* or *Hidden Volume* orders. The *Hidden Volume Trader* inserts orders with the price and volume decided in the same way as the traders above but also has an open volume smaller than it's volume. With a small probability the *Hidden Volume Trader* will attempt to enter a smaller volume than it's open volume, expecting an error. The open volume is the visible volume and is the volume that other market participants will see.

Stop Loss Trader

The *Stop Loss Trader* acts as a regular buy or sell trader except that it's orders are stop loss orders, meaning that the inserted order is suspended until the order is triggered by an event such as a particular ask price, bid price or last traded price. See chapter two for a description of different order types. The price and volume of the stop loss order is determined in the same way as above.

Market Operator

the *Market Operator* can trade halt order books and lift trade halts from order books. The order book is chosen randomly and the *Market Operator* will often

try to trade halt already trade halted order books and lift trade halts from order books that are not trade halted, expecting an error message. The *Market Operator* can also add and remove members and users. What action the *Market Operator* performs is decided pseudo-randomly.

5.3.2 The Oracle

The oracle is implemented as an actor. Trader actors send their orders to the oracle as well as the to trading system so the oracle can compare orders, from the traders to orders from the system in order to make sure that they are identical. When orders are matched, the oracle can compare the matched orders to the orders it has received from the traders to confirm that the traded orders actually have been inserted by the traders. The oracle also makes necessary boundary value tests. The following boundary value tests are performed:

Price not negative:

The price of an order is not negative.

Volume not negative:

The order quantity is equal to or greater than one.

Volume correct:

The volume of a trade is not greater than the volume of either of the orders matched in the trade.

Price correct:

The price of a trade is not higher than the price of the matched bid order or lower than the price of the matched ask order.

Other checks performed by the oracle are:

Traded order exists:

The orders matched in a trade have been inserted. The unique order-id is used.

Order not canceled:

The orders matched in a trade have not been canceled.

Order exists:

A canceled order existed prior to being canceled.

Order not duplicate:

An inserted order did not exist before being inserted.

oracle connected to system:

The oracle has received a *TAX heartbeat* in the last thirty seconds. This is a message from the system sent every 5 seconds.

User exists:

A removed user existed before being removed.

Member exists:

A removed member existed before being removed.

Member not duplicate:

An added member did not already exist.

User not duplicate:

An added user did not already exist.

Trade-halted order book not traded:

A trade was not made in a trade-halted order book

If the above requirements are not met, the oracle will return an error message and write the error to a log. The oracle was tested by changing the boundary values: a negative price was set to a price under 10, negative volume a volume under 2, A larger volume traded than bid was triggered if the volume was equal and so on.

5.3.3 Coverage

In order to measure coverage I will take a look at what transactions are used. One could use UML use cases instead but since I have a detailed list of transactions available but no use cases, the transactions are studied. The transactions used are listed in table 5.1. Even if only approximately 19.5% of the transactions are covered, the most important ones, in the sense that they are used often and failure will cause critical errors, are covered. These include OrderInsert, OrderUpdate and OrderCancel. 19.5% is the percentage of available transactions that where used. Many transactions can be used in several ways but this measurement gives a rough estimate of the coverage.

Actor	**Transactions**
Trader (This includes all traders)	TaxLogon, TaxLogout, TaxPrelogon, TaxSnapshotSubscribe, TestMessage, MassUpdate, OrderCancel, OrderInsert, OrderUpdate
Market Operator	TaxLogon, TaxLogout, TaxPrelogon, TaxSnapshotSubscribe, TradeHalt, AddUser, AddMember, RemoveUser, RemoveMember
oracle	TaxHeartBeat, TaxLogon, TaxLogout, TaxPrelogon, TaxSnapshotSubscribe
Not Implemented	62 transactions, mainly market management transactions

Table 5.1. Transactions covered by actors

5.4 Results and Analysis

5.4.1 Simulations

A simulation was run using traders, stop-loss traders, hidden-volume traders and market operators. The market operators trade halted, lifted trade halts and added and removed users/members. Three bugs were found. One resulting in a user remaining logged on and trading after being removed, one causing the TAX-server to crash and one causing the Common Data Server to crash.

The results were analyzed by measuring the Mean Time Between Failure(MTBF) and Mean time To Failure(MTTF). In order to measure MTBF, discovered bugs must be removed between each test. Since this couldn't be done fast enough, the functionality in the test that caused the error was removed instead. For example, when an error was found that crashed the Common Data Server(CD) due to an actor attempting to add an already existing member or user, the methods for adding and removing members and users were removed from the test. The amount of transactions sent to the system from the test was measured. The following bugs were found:

Sent Transactions	Error Description
210	Removed user remains logged in and continues to trade normally
2414	TAX crashes due to repeated and failed log on attempts by an already logged on user.
295571	CD crashes due to attempt to add already existing member
838746	Test complete

Table 5.2. Faults found by simulation

5.4.2 MTBF of the System Under Test

Mean Time Between Failures was calculated from the results above. Figure 5.2 shows the number of failures as a function of the number of sent transactions. It is

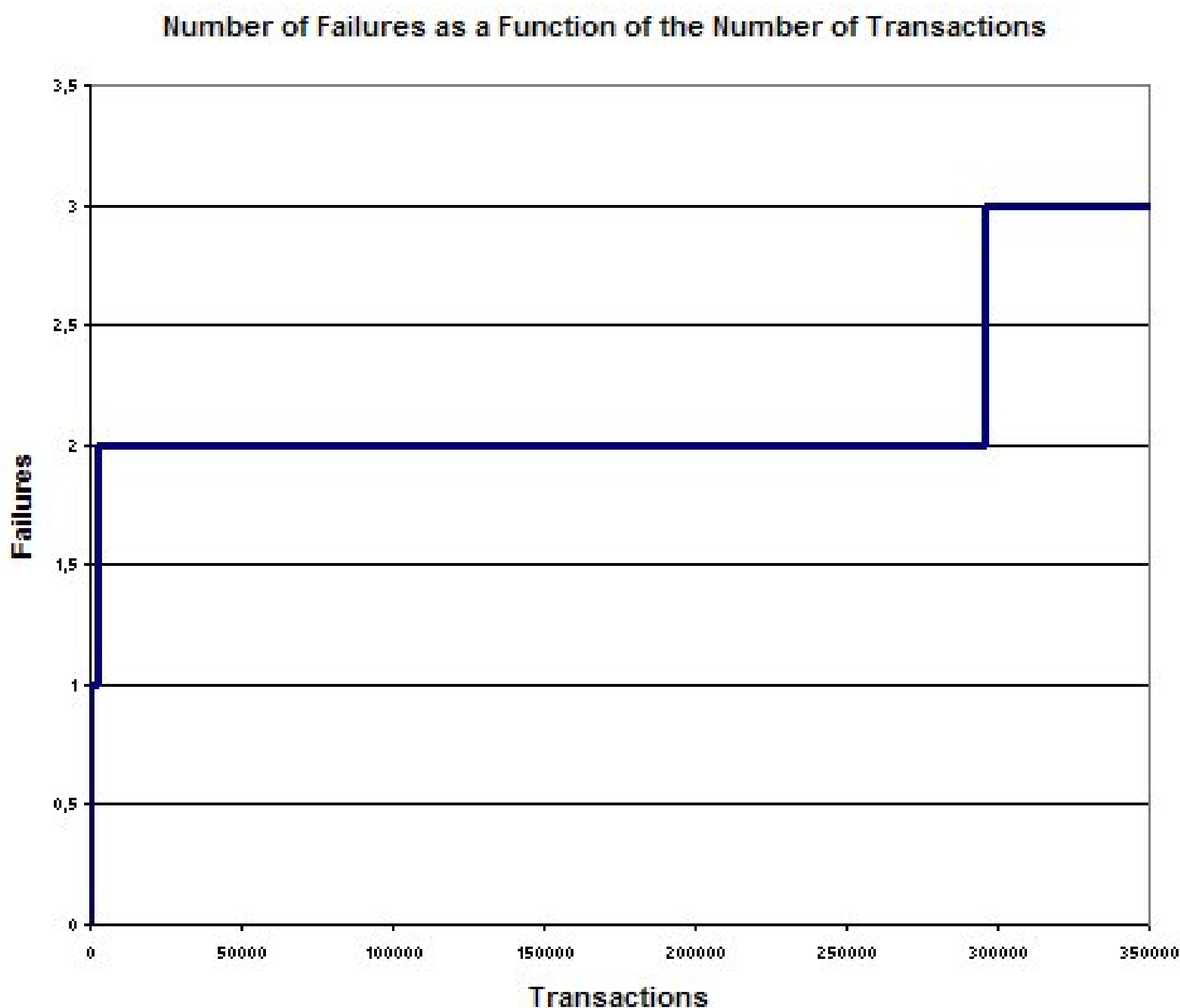

Figure 5.2. The number of faults as a function of time

obvious from the graph that the failure rate decreases with time when discovered bugs are removed. An approximation of the MTBF is difficult to obtain from such a small data sample but a very rough estimate is that in the first 2414 transactions,

when two failures have occurred, the MTBF is approximately 1207. Later in the simulation there are no errors and the MTBF can be considered small.

5.4.3 Bugs Discovered in the Common Data Server

First a user(trader) was added. The added trader was logged on to the system and started trading. When the user was removed by one of the market operators, the first bug was discovered. Although removed from the database, the trader continued to trade with other traders and was not logged off.

The market operators continued to add and remove members and users, frequently attempting to add already added users and remove non-existent ones, including the logged in non-existent trader. This test was run several times and each time it caused the Common Data Server(CD) to crash. The crash was caused by the market operators trying to add an already existing user. In order for the crash to occur the simulation needed to be run for a while and the first bug needed to be exploited.

This is a good example of the kind of low-frequency bugs that random tests can find.

5.4.4 The TAX Bug

Another bug discovered by the random tests was when already logged on users were allowed to attempt to log on again. A script that attempted to log on each user repeatedly until successful was run. This cause the TAX to crash after only a few minutes.

Mean Time To Failure(MTTF) can be measured by running a test and measuring the time from a failure free state to failure without removing a bug between each run. The test is run until the system crashes. This was done for the TAX-crashing bug with the results shown in table 5.3.

5.4.5 MTTF for the TAX-crashing Bug

Mean Time To Failure(MTTF) was calculated, from the results in table 5.3, for the TAX-crashing error. The MTTF for the TAX when running the simulation was calculated to be 11638 transactions with a standard deviation of 13722. However, when we examine the results (see figure 5.3) it is clear that the number of transactions is concentrated to three areas: around 2300 around 9000 and around 35000. This implies that there might be several bugs causing the TAX to crash and this must be investigated. In order to check if something other than users attempting to log on over and over again was causing failures the test was run without the actors attempting to log on more than once. No failure was found during this test.

Number of transactions to TAX-server crash
2329
2326
2367
38235
37040
39801
36536
9341
8747
2326
7563
2699
7576
8968
2336
9643
3693
9054
9618
2326
2308
2327
2314
2327
34353
3421

Table 5.3. MTTF for a system containing the found TAX-bug when running the simulation

Since some actors attempted to log in over and over again and some were logged in and able to trade, the number of transactions for each actor varied greatly. In order to get a better measurement of how long it takes for the test to crash the TAX, the actors were not allowed to do anything other than attempt to log on over and over until they succeeded. The results are shown in table 5.4.

The MTTF is then calculated to 2415.1 with a standard deviation of 8.77. It can be assumed that the deviations in the previous test run were caused by different types of transactions taking different amount of time. When only log on transactions are performed it becomes obvious that the bug is actually caused by actors attempting to log in over and over again.

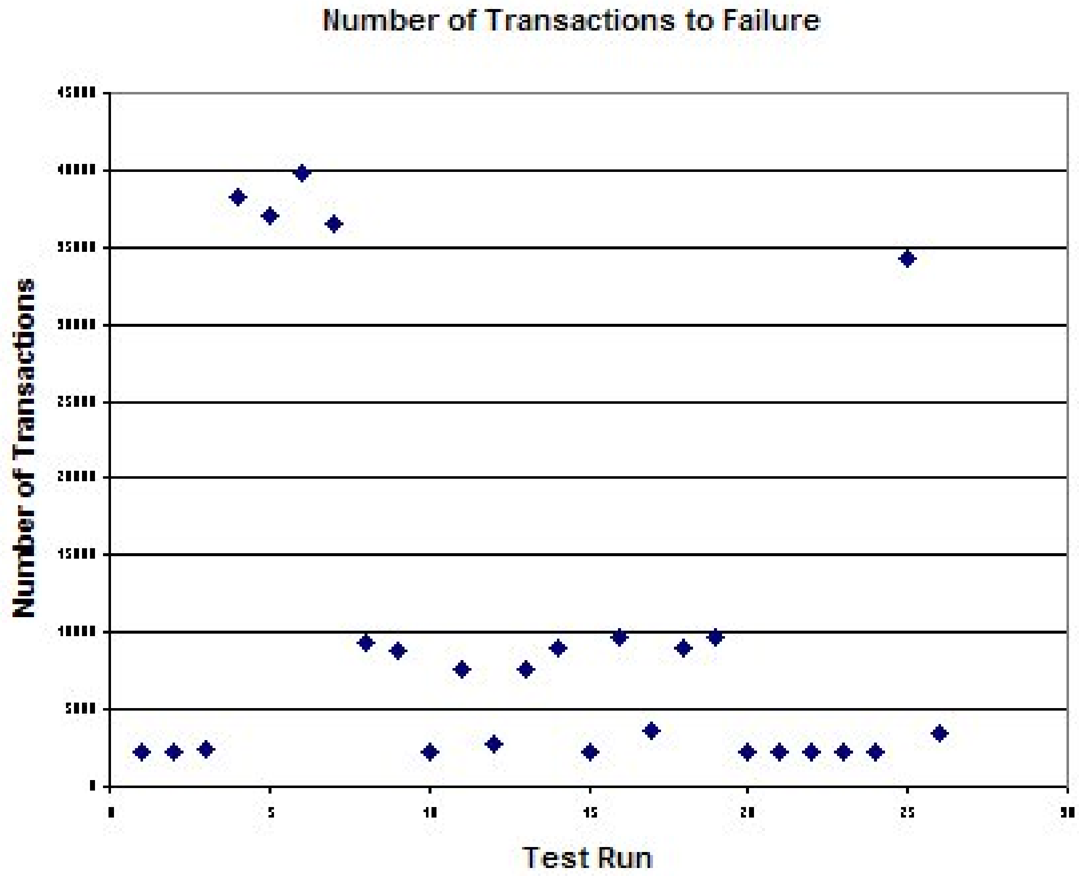

Figure 5.3. The number of transaction before failure for each test run

Number of transactions to TAX-server crash
2416
2422
2413
2411
2405
2410
2417
2436
2413
2408

Table 5.4. Number of transactions before TAX failure

5.4.6 Oracle Analysis

Determining whether the oracle was effective and whether the right oracle was chosen, is difficult. The main difficulty arises from the small size of the project. Had the oracle had more functionality and checked for more errors, perhaps it would have found bugs. The faults that occurred during my simulations were detected by crashes or by manually observing the output of the system. It is possible that another oracle strategy would have found more errors. A no oracle strategy would have found the same amount of errors with less work. However, the possibility that a heuristic oracle will find errors if used in larger random tests of trading systems cannot be ruled out.

Chapter 6

Discussion

In this master's project a previously tested system, currently in production, was used. That any bugs at all were found can be seen as evidence of random tests being effective in finding bugs that otherwise would have gone undetected. The errors found are not very likely to occur regularly in a trading system, however they are all critical faults, most of which would cause the system, or part of the system, to crash. This thesis aimed to test the hypothesis that random tests can find errors that cannot be found by regular automatic and manual tests. Considering that several bugs were found, that had gone undetected by other tests, it appears that random tests do find such bugs.

One may argue that the bugs found in this test could have been found by other automatic tests. For instance, the TAX-crashing bug is best found simply by allowing several actors attempt to log in to the system over and over again without any randomness involved at all. However, previous tests did not find this error and if random tests had not been used, the idea to let several actors attempt to log in over and over again, might not have come up.

The errors found in this project were discovered either through observation or by servers crashing. The oracle designed and used in the project did not find bugs. The reason for this could be that the oracle only checked already tested functionality, meaning that a better oracle would find more errors. One might question the use of a heuristic oracle since it can miss many errors. In this project, using a no oracle strategy would have been equally effective.

One main advantage of random tests is that they are cost-efficient. Creating a random test tool that found the bugs mentioned in this thesis took approximately 320 hours. How long would it take to implement other automatic tests with the same coverage? According to Lars Wahlberg, Test Strategist at Cinnober, the two methods cannot even be compared. If one lets a random test run for a long time without human intervention, it will cover many transaction sequences that never

would be covered by automated functional tests. Finding crashing bugs is difficult with other tests and in comparison barely takes any time at all with random tests. Random tests in combination with other manual and automatic tests appear to be efficient. But what would the result be if only random tests were used? Through random testing many bugs can be found quickly but analyzing them takes longer than it does with other automatic test tools. If only random tests were used it is possible that many bugs would go undetected and that it would end up costing as much or more than only using regular automatic tests. Random tests are efficient but other tests such as tests based on boundary value analysis or equivalence class partitioning, are still necessary.

The system tested in this project was ”basically completely tested” when the random tests were run, yet it didn't take long for faults to appear. That some bugs were found by random tests might not be enough to prove that random tests are effective in finding low-frequency bugs, but it certainly implies that they do.

6.1 Conclusion

The hypothesis tested in this thesis is that random tests are effective in finding low-frequency bugs that otherwise would go undetected by testers. Several tests were run and several bugs were found. The expected number of faults in this test was zero. A total of five bugs were found by the random tests performed. In 500000 transactions four bugs appear. Random tests do find bugs that other tests do not find.

6.2 Further Work

There is much to be said about random tests. The test used in this project could be developed into a larger random test which could be used for testing all systems created by Cinnober. Further studies of random tests in trading systems should be done as random testing has proved to be an effective tool in discovering errors.

A master's project focusing only on the creation of a test oracle could be useful as a next step toward finding a more effective oracle.

References

[**Bach 2001**], Bach, James *What is Exploratory Testing?*, www.stickyminds.com 2001

[**Bach 2003**], Bach, James *Exploratory Testing Explained*, http://www.satisfice.com/articles/et-article.pdf, 2003

[**Fu 2003**], Fu, Rui *Automatic Test Suite Generation Based on Test oracles*, The University of Western Ontario, Department of Computer Science, London, Ontario, 2003

[**Graham 2007**], Graham, Dorothy; Van Veenendaal, Erik; Evans, Isabel; Black, Rex *Foundations of Software Testing*, Thomson Learning, London, 2007.

[**Gundemark 2005**] Gundemark, Johan, *Random tests in a Market Place System*, Uppsala University, Department of Information Technology, Uppsala, Sweden, 2005.

[**Hoffman 1999**], Hoffman, Douglas, *Heuristic Test oracles*, STQE Magazine March/April 1999, (pages 29-32).

[**Hoffman 2000**], Hoffman, Douglas *Mutating automated Tests*, STAR East, 2000

[**Hoffman 2001**] Hoffman, Douglas, *Using Oracles in Test Automation*, Software Quality Methods, http://www.softwarequalitymethods.com/Papers/Auto

[**Johansson 1996**] Johansson, Erik, *Safety-Critical Control Systems- The Challenge of Migrating from Hardware to Software*, Industrial Control Systems, School of Electrical Engineering and Information Technology, KTH, Royal Institute of Technology, Stockholm, Sweden, 1996.

[**Kaner 2000**], Kaner, Cem *Architectures of Test Automation*, http://www.kaner.com/pdfs/testarch.pdf, 2000

[**MIL 2000**], MIL-STD-882D, Department of Defense, *Standard Practice for System Safety*, United States Department of Defense, 2000.

[**NASA 2004**] *NASA Software Safety Guidebook*, 2004.

[**Schauer 2006**], Schauer, Philipp Martin *Market Architecture of The Largest Stock Exchanges*, Leopold-Franzens-Universität Innsbruck, Institut für Banken und Finanzen, Fakultät für Betriebswirtschaft, 2006

[**Sellberg 2003**], Sellberg, Lars-Ivar, *MEXUS-main design ideas v1.3*, 2003

[**Spillner 2006**], Spillner, Andreas; Linz, Tilo; Schaefer, Hans, *Software Testing Foundations*, dpunkt.verlag, Heidelberg, 2006

[**Sthamer 1995**], Sthamer, Harmen-Hinrich, *The Automatic Generation of Software Test Data Using Genetic Algorithms*, University of Glamorgan, 1995

[**TE 2007**],*TRADExpress Trading Engine Overview*, Cinnober Financial Technology, 2007

[**Wååk 1978**], Wååk, Olof, *Driftsäkerhetsteknik- Introduktion, begrepp och samband*, Systecon AB, 1978

[**Zhan 2002**], Zhan, Yuan, *Automated Test-Data Generation*, University of York, Department of Computer Science, 2002