

# Random Tests in a Market Place System

---

Johan Gundemark

# **Abstract**

As transactions at exchanges around the world constantly increase the importance of reliable market place solutions grows. Normal test are often not enough to meet the high demands on availability from customers and traders world wide. To further develop the test techniques random test have been introduced. These tests use random parameters to create input to the system. Low frequent combinations of transactions which cause the system to fail can be found and removed. Faults that otherwise would go undetected.

This master thesis focuses on random tests in general and gives an explanation of the different techniques used today. It also includes a practical study where faults are introduced in a small system. By injecting faults and running simulations it is possible to get an estimate of the reliability of a system.

# Sammanfattning

Handel på börser runt om i världen ökar konstant och för att garantera kundernas säkerhet krävs att den mjuk- och hårdvara som används möter de höga krav som ställs. Börsmarknaden är konkurrensutsatt och klarar man inte av att leverera ett väl fungerande system kommer kunderna att använda sig av alternativa lösningar. För ett börssystem är det extra viktigt med hög tillgänglighet då ett stopp på endast ett par timmar innebär betydande förluster ekonomiskt och dessutom ger det företaget som utvecklat systemet badwill. Därför är tester och verifiering en viktig del av arbetet.

Detta examensarbete fokuserar på slumpmässiga tester. Rapporten innehåller dels en teoretisk del som tar upp hur slumpmässiga tester konstrueras, samt en kort studie hur de kan användas i praktiken. Arbetet är utfört på OMX Technology i Stockholm som bl.a. utvecklar den plattform som används vid handel på Stockholmsbörsen.

För att försäkra sig om att produkterna som levereras av OMX Technology klarar av de högt ställda kraven genomförs utförliga tester. Funktionella tester verifierar att systemet beter sig på ett korrekt sätt och icke-funktionella tester kontrollerar t.ex. att systemet klarar av extremt hög belastning under en kortare tidsperiod.

Som ett komplement till de ordinarie funktionella och icke-funktionella testerna har slumpmässig testning tagits fram. Vid dessa tester används olika modeller, kallade aktörer, som oberoende av varandra tar en speciell roll i handelssystemet. En aktör kan t.ex. endast lägga en viss typ av ordrar medan en annan endast köper vissa förutbestämda aktier. Aktörernas handelsmönster bestäms av parametrar som sätts innan testerna startar. När testerna sedan utförs agerar aktörerna oberoende av varandra, liknade hur ett handelssystem fungerar i praktiken. På detta sätt kan komplicerade scenarion som leder till fel upptäckas, fel som inte de ordinarie testerna skulle fånga upp.

Slumpmässiga tester kan även utföras på andra sätt. Huvudidén med denna typ av tester är att inget test är exakt det andra likt. Genom att slumpa fram vissa parametrar inom ett givet intervall kan i praktiken ett oändligt antal testfall skapas. Den output som genereras kontrolleras sedan mot inputen med hjälp av ett så kallat orakel. Oraklets komplexitet kan variera från att bara upptäcka spektakulära krascher tiol att veta exakt hur utdatan ska se ut i förhållande till indatan.

Genom att injicera fel i ett system och sedan göra tester med slumpmässig indata kan man få ett mått på systemets tillförlitlighet och en uppfattning om kvalitén på de tester som utförts. Detta görs genom att jämföra förhållandet mellan antalet injicerade fel och antalet upptäckta fel. Hittas ett stort antal av de injicerade felen kan man anta att testerna täcker upp en stor del av systemets potentiella indata medan ett resultat där endast ett fåtal fel hittas tyder på att indatan bör ses över.

Den praktiska studien som utförts bygger på en felinjiceringsmodell. Istället för att utföra testerna på ett börssystem, som är oerhört komplext, har ett enklare system använts. Den modell som valdes var en schacksimulator med möjlighet att låta två artificiella intelligenser spela ett parti. Ett schackparti har vissa slående likheter med hur handel i ett börssystem går till i det att enkla drag/transaktioner leder till mer komplicerade scenarion.

Schacksimulatorens modifierades så att det tillät partier utan mänsklig medverkan. Ett visst slumpmoment infördes även när de artificiella intelligenserna skulle räkna ut det optimala draget för att undvika att alla partier såg likadana ut. Dessutom lades funktioner till som markerade om vissa fördefinierade mönster uppträdde i schackpartiet. Händelser som med viss sannolikhet kan inträffa under ett normalt parti, t.ex. att en häst slog en annan häst eller att en bonde flyttades tre gånger i rad. Dessa händelser kan även ses som fel av olika allvarlighetsgrad som på något sätt påverkar systemet så att det inte fungerar som önskat.

Ett antal schackpartier spelades sedan och för varje parti registrerades vilka av de fördefinierade mönstren som uppträdde i det nuvarande partiet. Genom att analysera vilka händelser/fel som inträffade fås en uppfattning om hur tillförlitligt systemet är. Man kan t.ex. få svar på om systemet har klarat av den önskade tillförlitligheten samt om det har fler eller färre fel än väntat.

# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>7</b>
1.1	Background .....	7
1.2	Assignment .....	8
1.3	Disposition .....	8
1.5	Acknowledgements .....	8
<b>2</b>	<b>The Trading System .....</b>	<b>9</b>
2.1	Trading .....	9
2.2	The Trading System .....	9
2.3	Market Hierarchy .....	9
2.4	Trading States .....	10
2.4.1	Submarket states .....	10
2.4.2	Order book states .....	11
2.5	Transparency .....	11
2.5.1	Order Transparency .....	11
2.5.2	Counterparty Info Transparency .....	11
2.6	Order Classes .....	12
2.6.1	Round Lots .....	12
2.6.2	Odd Lots .....	12
2.6.3	Block Lots .....	12
2.7	Order Types .....	12
2.7.1	Single Orders .....	12
2.7.2	Contingency Orders .....	12
<b>3</b>	<b>The Testing System .....</b>	<b>14</b>
3.1	Creating test cases .....	14
3.2	Evaluation .....	15
3.3	Simulations .....	15
3.4	Simulations evaluation .....	15
<b>4</b>	<b>Random Tests .....</b>	<b>16</b>
4.1	Overview .....	16
4.2	The Input .....	18
4.2.1	Distribution .....	18
4.2.2	Domain .....	18
4.2.3	Size .....	18
4.2.4	Trading simulations .....	19
4.3	The Oracle .....	20
4.3.1	No Oracle .....	23
4.3.2	True Oracle .....	23
4.3.3	Consistent Oracle .....	24
4.3.4	Self-Referential Oracle .....	24
4.3.5	Heuristic Oracle .....	24
4.4	The Output .....	25
4.4.1	Reliability .....	25
4.4.2	Reliability models .....	26
4.4.3	Reliability assessment .....	28

<b>5 Implementation and Results.....</b>	<b>29</b>
5.1 Picochess .....	29
5.2 Modifications .....	30
5.3 Simulations.....	30
5.4 Analysis.....	30
<b>6 Discussion.....</b>	<b>35</b>
6.1 Issues .....	35
6.1.1 Platform.....	35
6.1.2 Simulations.....	35
6.2 Further Work.....	35
6.3 Summary .....	36
<b>References .....</b>	<b>37</b>

# 1 Introduction

---

## 1.1 Background

Exchanges around the world have seen a rapid increase in both turnover and revenue in the last decades. With the introduction of online trading the possibility for a private trader is almost identical to the big brokers that traditionally handle the majority of transactions. This increase in trading forces the exchanges to manage large amounts of information in a short period of time, resulting in higher demands on the exchanges and the trading systems they provide.

As many of the large exchanges around the world now run as private corporations they are put under competition. If the trading system provided does not meet the demands of the customers, both brokers and the companies being traded, they will take their businesses elsewhere. It can take years to build up a reputation as a fine exchange but only days to ruin it. Because of the amounts of money being handled at an exchange it is second on the demands of reliability and availability only to life-crucial systems such as e.g. aviation. A small error that causes trading to stop for only a few hours is extremely expensive.

Consequently it is important for an exchange to have a solid test environment with well-defined test methods. Both hardware and software are put under the lens to ensure that the demands of the customers are met. Hardware can be duplicated and therefore offer redundancy if one of the mirrored system should fail. With software it is too expensive to develop several independent systems. This means that the exchange has to rely on its software tests to find the potential faults that could result in major failures of the trading system.

OMX Technology is part of OMX. The company provides technology and services for financial markets around the globe. Among the customers you find exchanges as the Stockholm Stock Exchange (SSE) and the American Stock Exchange (AMEX) as well as market participants such as JP Morgan. [17]

To ensure that the products delivered by OMX Technology meet the high demands from the customer thorough tests of the system are carried out. Both functional and non-functional tests are performed. The functional tests verifies the correctness of the system functionality and the non-functional tests focuses on e.g. the ability to process large number of transactions during a short time period, also known as stress testing.

To make the tests even more efficient OMX Technology has developed a number of trading models, called actors, which simulates random trading in the market. This way trading sequences that will cause the system to fail can be found, faults that will not be captured with the ordinary functional tests. This method has proved to be useful in finding low frequent yet serious faults. [5]

## **1.2 Assignment**

This work is a thesis for the Master of Science Program at the department of Information technology at Uppsala University. The work was done for OMX Technology in Stockholm. Supervisor at the University was Arne Andersson and supervisor at OMX Technology was Lars Wahlberg.

The goal of this thesis is to further develop the random tests at OMX Technology. The work is divided into two parts. One theoretical part that generally describes the concepts of random testing. One part where a small practical study is carried out. The study focuses on fault seeding and the possibility to get an estimate of the reliability of a system. By injecting faults and running a number of simulations it is possible to estimate how many unknown remaining faults the system has as well as getting an estimate of its reliability.

## **1.3 Disposition**

Chapter one gives a short introduction to the subject as well as a description of the assignment.

Chapter two includes a description of trading and how a modern trading system works. It gives an understanding of the different types of transactions performed at an exchange today.

The next chapter, chapter three, explains how tests are performed at OMX today. Information about the different testing techniques and the way test are evaluated are given.

Chapter four explains the fundamentals of random testing. Issues such as the complexity of oracles and the input and output from a system under test are discussed.

The implementation of fault seeding in a system environment and the results of this study are discussed in chapter five.

The last chapter, chapter six, includes issues during the development and ideas about future work.

## **1.5 Acknowledgements**

First and for most I would like to thank my supervisor at OMX Technology Lars Wahlberg for giving me the opportunity of doing this master thesis and for good feedback along the way.

I also would like to thank Uppsala University for setting up the details and helping out with some administrative issues.

## 2 The Trading System

---

### 2.1 Trading

Exchanging goods and services has always been a way for people to acquire things that they cannot produce themselves. The trade started in spontaneous marketplaces but soon moved to more permanent trading locations in the cities. A farmer could make an agreement with a merchant that he would get an in advance agreed price for his goods. This way the farmer did not have to worry about the trade and could focus his work on what he did best, farming.

To further develop the trading of goods and services exchanges with members were created. Only the members were allowed to trade at the exchange and all trading had to go thru them. The members had certain rules to follow to avoid being excluded from the exchange. This way fairness was guaranteed. Today's modern exchanges are based on these ideas although the goods have been replaced with financial instruments such as stocks and options. [1]

### 2.2 The Trading System

The trading system information is taken from reference [4] if not stated differently.

OMX Technology provides the software and hardware needed to operate and run a complete market place system. The hardware offers the basic platform and the software controls the trading and supports the use of different applications. This suite of applications is used for data administration, operations supervision, trading control and market surveillance. [2] The members also use applications to connect to the market place system. These applications are OMX developed, commercial trade applications or an own computer system with trading functionality. [3]

### 2.3 Market Hierarchy

- Exchange

An exchange is a marketplace containing one or several markets. These markets are divided into a number of smaller submarkets that contain different order books.

- Market

A market is a logical part of an exchange. The market is often divided into one or more submarkets with different trading rules.

- Submarket

A submarket is a part of a market that consists of a logical group of order books. Within a certain submarket the order books are generally of the same type, e.g. stocks

or bonds. The order books are arranged as submarket to be able to take advantage of the fact that they share the same trading rules, e.g. the same trading hours etc.

- Order Book

The Order book is where the actual orders are stored and where trades are concluded. It is defined by three parameters: Order book ID, trade currency and submarket.

The major advantage of this structure is its flexibility. Individual markets can be combined to create new market models and a number of parameters can be configured to form the required rules for an individual submarket.

## 2.4 Trading States

During a trading day the order books and submarkets go thru a sequence of different states. The schedule for these sequences can be uniquely configured for every submarket. Each of the states is associated with a set of allowed operations and a level of transparency.

### 2.4.1 Submarket states

A trading schedule for a submarket could be setup as follows:

- Closed
- Initiating
- Pre Trade
- Trading
- Trade Termination
- Post Trade
- Closed

In the closed state the submarket is closed and the order books assigned to it are closed as well.

During the initiating state the access to the market is restricted. The members can retrieve static data but it is not possible to enter orders.

The preparing trade state allows orders to be entered but no automatic matching is performed.

In the trading state orders are matched against each other on a continuous basis.

During the short trade terminating state the closing index is set. This allows users to get an overview of the market when the trading was halted. No order entering or cancellation is allowed.

The post trade state is similar to the preparing trade state. Order maintenance is allowed but no matching is performed.

The most common trading state is continuous trading. This is the type of trading most people think of when they are asked to describe trading at an exchange. Orders are matched according to price-time priority. There are several other market models, e.g. different types of auctions but the thesis will henceforth concentrate on continuous trading only.

### **2.4.2 Order book states**

The trading schedule for an order book often follows the schedule of the submarket it belongs to. However it is possible to control the status of single order books if special circumstances arise. When the order book is open it can be put into the following states:

- Trading halt
- Technical stop

Trading halt temporary enables the possibility to enter orders into the order book. It can not only be applied to a single order book but also to a certain issuer or instrument, in which case trading in related order books are halted as well. A variant of trading halt is matching halt where off exchange reporting is allowed but no automatic matching is performed.

If for some reason an order book becomes unavailable due to technical reasons a technical stop is made.

## **2.5 Transparency**

The level of transparency indicates to what extent order and trade information is public. There are two different types of transparency: order transparency and counterparty info transparency.

### **2.5.1 Order Transparency**

Order transparency is defined per submarket, state and order class. The following levels of transparency are available:

- None
- Market-By-Order (MBO)
- Market-By-Level (MBL)

Market-By-Order is the highest level of transparency. Each individual order including price and volume is shown in the order book.

Market-By-Level represents a view where all orders on one price are aggregated and shown as a total volume for that level.

### **2.5.2 Counterparty Info Transparency**

Counterparty information is defined per submarket and can be set to show no or full information. It is also possible to configure the submarket to make orders of a certain size anonymous.

## **2.6 Order Classes**

Orders are classified as round, odd, or block lot depending on their size. This partition into different classes is to reduce market impact of huge orders. Transparency level can be set per order class, which makes it possible to only show part of a large order.

### **2.6.1 Round Lots**

The round lot order must have a size equal or higher than the round lot size. This order type defines the current spread between buy and sell orders and the last paid price of the stock. Most trades are made as multiples of the round lot size.

### **2.6.2 Odd Lots**

Odd lot orders are smaller than the round lot size. The system constantly tries to group and match odd lots against other odd lot and round lot orders.

### **2.6.3 Block Lots**

Block lot orders must have a size equal or greater than the block lot size. This order type is used for reducing the market impact of huge orders by setting a low transparency level. Block lots are matched against other block lots and round lots.

## **2.7 Order Types**

### **2.7.1 Single Orders**

The following list is examples of attributes that can be applied to single orders. It is not complete but includes the most common attributes.

- **Fill-Or-Kill.** When the order is entered into the market it must be matched completely otherwise it is deleted.
- **Fill-And-Kill.** The amount of the order is filled up as much as possible to the stated price. The remaining part of the order is deleted.
- **Hidden Volume.** Only a portion of the total order amount is shown in the market. When this portion gets matched the open amount is refilled from the total amount and this continuous until the whole amount has been matched.
- **Valid Until.** The order is only valid until a specified date or time.

### **2.7.2 Contingency Orders**

The price and volume of a contingency order is related to the price and volume of other orders in the market. These are examples of available contingency orders:

- **Stop-Loss orders**
- **Linked orders**
- **Combination orders**

Stop-Loss orders change their price automatically when a certain price condition in the market is met. The idea is to trigger the order when the change occurs, for example to sell a stock when the price reaches a previously defined level.

Linked orders provide functionality to enter more than one order under the condition that you want to buy for example either 500 units of stock A, or 500 units of stock B, or a combination of the two stocks. The linked order corresponds to a number of single orders with an exclusive OR-condition. This allows the user to control the risk since he can only end up making a transaction in one of the order books.

Combination orders resemble linked orders but the OR-condition is replaced with an AND statement. For example a user wants to buy 300 units of instrument A and 500 units of instrument B. A deal will only be made if both conditions are met. Combination orders make it possible to use different complex trading strategies.

## 3 The Testing System

---

The information in this chapter is taken from reference [5].

### 3.1 Creating test cases

OMX uses a test tool that facilitates the configuration and creation of test cases. The tests are divided into two parts. Trivial tests, e.g. mandatory field should be correct, and more advanced simulations.

The trivial tests are to a very high degree automated. Manually specifying these test cases would be time consuming and boring, which increases the risk of typos.

The straightforward way of designing trivial test cases is to create a script that performs the necessary transactions and evaluates the result. The problem with this is that each transaction often consists of 20-60 different mandatory fields. For a typical test case however only 5-10 fields are of importance. This does not allow the remaining fields to be left empty, they still have to be setup with correct information to make the test possible.

When designing test cases this leads to a lot of unnecessary work for the programmer. Consideration has to be taken to all mandatory fields, not only the ones relevant in the specific test case. Maintenance of the tests also increases with the mandatory fields. Small changes in the trading rules would require that all test cases are reviewed. To solve this problem a high level of abstraction is used when the test cases are created.

A generalized test case could be explained as follows:

- Test case environment setup. The test case must be executed in an environment that supports the functionality being tested.
- Stimuli generation. Creation and input of transactions.
- Evaluation of the generated transactions.

In order for a test case to be executed it has to be run in an environment that supports the tested functionality. Instead of manually selecting and hard coding the necessary data, e.g. the order book, the programmer requests an order book with a number of properties. The test system will then provide an order book that meets these predefined requirements.

When the setup of the environment is done the stimuli needs to be generated. No volumes and prices are hard coded. The programmer requests suitable parameters according to the order book and the system provides the data. This works in a similar way to the setup of the environment.

The programmer can if necessary always be more specific when designing the environment and creating the test cases by setting specific mandatory fields manually. The advantage with this high level of abstraction is that the scripts become easy to create and maintain.

## 3.2 Evaluation

The evaluation of the test results are done in a similar way. The programmer can choose the appropriate level of abstraction. Examples of questions that can be applied to the test results are:

- Did the trade occur?
- Is the order book in the correct state?
- Which orders were cancelled?

If necessary it is possible for the test case author to examine each transaction field in detail manually. By using abstraction when analyzing the results the time consumption decreases significantly allowing the programmer to analyze a greater number of test cases.

## 3.3 Simulations

The possibility of large number of configurations and orders in an order book makes the test space very large. From a practical purpose it can be considered infinite, hence it is not possible to cover all scenarios with regular test cases.

The approach is to complement the normal test cases with trading simulations. A simulation that finds an error in the system results in a review of the test cases for that specific functionality. If necessary extra test cases are added to the test portfolio.

The simulations use different actors that have a specified role in the market system. One actor can e.g. only cancel orders whereas another actor only enters stop-loss orders. The actions of an actor are associated with a probability factor that determines how often the action should be executed. Order volume, price etc. is randomized according to preset parameters. Normally a typical actor has 10-30 actions and 2-3 randomized parameters.

When a simulation is run a mix of actors and probabilities are configured. Each actor works on its own without knowing the actions of the other actors, similar to a real market place system.

If a fault is found it is important to have the ability to rerun the entire scenario that led to this error. Therefore a given seed is used when all randomized values are calculated.

## 3.4 Simulations evaluation

The advantage of trading simulations is the possibility to generate complex trading scenarios that would be difficult to catch with regular test cases. The drawback is that these simulations are hard to evaluate. The main area for the simulations is to ensure that a certain transaction sequence does not cause the system to crash. Together with regular test cases simulations is powerful to ensure a secure system.

Trading Simulations are further discussed in chapter 4.2.4.

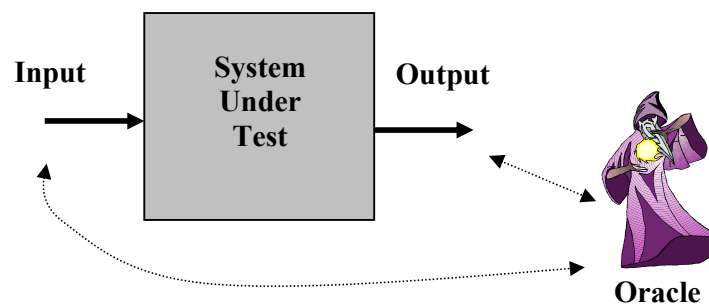
# 4 Random Tests

---

## 4.1 Overview

Automated tests have been used to examine and analyse computer software during the last decades. As test routines improve the use of more sophisticated test tools grows and one important area of interest is random tests (also known as Monte-Carlo tests). [6]

The idea with random tests is to feed the system under test with random input from a pre-selected input domain. This makes it possible to control the boundaries off the chosen data without restricting it to any specific values. A so-called oracle then analyzes the output and decides if it was correct or not. The oracle can have different degrees of complexity, from very simple crash-only checks to an exact replica of the system. [6] This will be further discussed later on.



*Figure 1, Random Tests [6]*

When developing regular automated tests a wide range of different techniques are used. These techniques can be applied when designing random tests as well. A list of some of the most common practices that especially apply to random testing is presented below: [7]

1. **Partitioning:** The input and output are partitioned into smaller subsets that can be treated in similar ways. Tests are then performed on each partition to verify correct functionality. Both valid and invalid input data can be tested this way.
2. **Boundary value analysis:** Design test cases for data close to the edge of partitions, faults are more likely to appear at these boundary values.
3. **Probabilistic testing:** Designed to test stability of the system rather than finding bugs. The system is fed with input data and after a limited time period it is possible to get an estimate of the failure rate of the tested system. These tests are important for safety-critical systems, which have to be extremely reliable.

4. **Error seeding:** By inserting faults in the code it is possible to get an estimate of the efficiency of the testing procedure. This is done by measuring the number of faults found versus the number of injected ones. An approximation of the quantity of remaining faults can be found obtained this way.

One major limitation of regular automated tests is that they become more and more unlikely to find defects as the number of times a test case is executed increases. [8] Normally test engineers who automate test cases run the test manually before implementing the automatic solution, meaning that the first time the automated test case is executed is not the first time that particular functionality is tested. One can see that this makes automated testing great for regression tests, (testing old functionality on a regular basis to make sure that it is compatible with new functionality) but it is very unlikely to find new bugs in young code. [8]

*Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason that stepping in someone else's footprints minimizes the chance of being blown up by a land mine.*

*James Bach, Test Automation Snake Oil, Windows Tech Journal, October 1996*

By randomizing the input data new tests are constantly created and although a majority of the test cases will not find defects in the system, every once in a while a specific sequence that evokes a serious error in the system will take place. By combining random tests it is possible to create more and more complicated test cases. This is useful as the system gets more stable and advanced tests become necessary. [9]

*The beauty of random testing (...) is that we can keep testing previously tested areas of the program (...) but we are always using new tests (making it easier to find bugs that we've never looked for before).*

*Cam Kaner, Architectures of Test Automation, August 2000*

It is important to see random tests as a complement to more traditional test approaches. It will never replace these tests but can, when used wisely, find complicated low frequency bugs that otherwise would have passed the test procedure undetected.

*It may not find a large number of bugs, but it is an excellent sanity check on the rest of your testing: If you are outperformed by random testing, you may have a problem on your hands. And I am always pleased with the high quality (albeit few in number) bugs that random testing manages to find.*

*James Whittaker, [11]*

There are three different challenges when designing a successful random test environment. [6]

1. **The Input** (distribution, auto-generation, domain of input values etc.)
2. **The Oracle** (complexity of verification, design etc.)
3. **The Output** (reduction of data, reliability and coverage measurements etc.)

These challenges will be discussed separately and different approaches and viewpoints will be considered.

## **4.2 The Input**

The input data is very important when creating random tests, or any tests at all. Without good input to the system the test case will not find a lot of errors. It does not matter how good the verification oracle is if the system is fed with bad input. [10]

A benefit that comes from defining input for a random test is that the tester gets knowledge about the system while creating the input. This knowledge is obtained when the system is analyzed, and a good way to come up with interesting testing ideas is to plan random tests and generate the necessary input. [11]

### **4.2.1 Distribution**

Feeding the system with realistic data can be a good way to measure stability. By doing this it is possible to supervise the system during normal conditions. Mean Time To Failure (MTTF) is frequently used when measuring the stability of a system. It indicates the average time from an initial failure-free state to a failure. If the time to repair a system can be disregarded compared to the time of operation the term Mean Time Between Failure (MTBF) can be seen as identical to MTTF.

A good way to obtain realistic data for a market place system is simply to repeat earlier days of trading. This way previously developed functions can be thoroughly tested in the new system before set into production. To test new functionality the input data must be produced in an alternative way since old trading days will not include the latest functionality.

By creating unrealistic data and feeding it to the system it is possible to produce series of transactions that normally would not happen during an ordinary day of trading. This makes it possible to find low frequent error that might have passed thru the function tests.

To achieve a successful random test strategy both distributions are important, realistic data to measure the stability of the system and unrealistic data to find low frequent bugs. [6]

### **4.2.2 Domain**

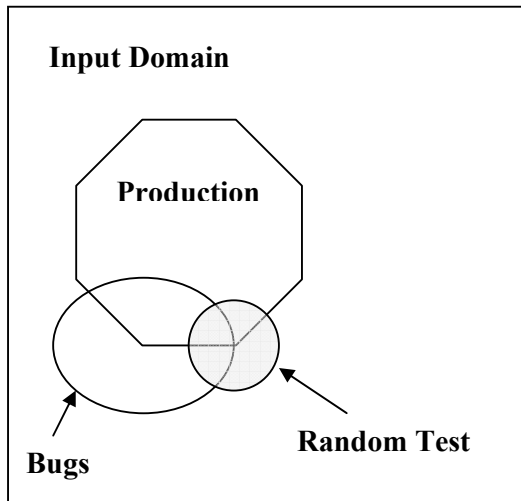
Another important aspect of random tests is the domain of input values. The input domain for a complex system is in reality infinite, but by choosing to send data from selected sub domains it is possible to do extensive testing with a limited number of test cases. Different parts of the code will be executed depending on the input domain. Because certain inputs will trigger certain parts of the systems functionality it is possible to cover selected areas of the code. Sending incorrect data to the system will test error handling. [6]

### **4.2.3 Size**

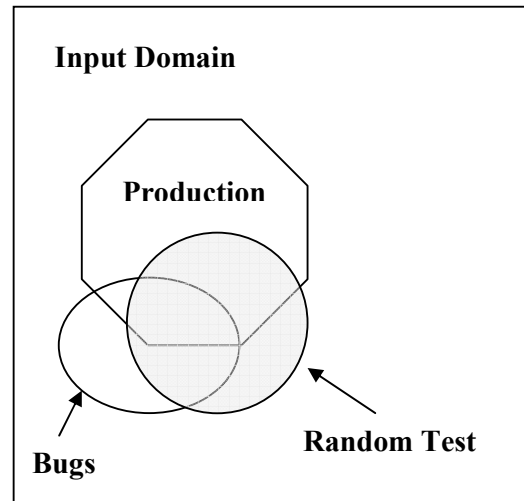
What size should random tests have, and how is it measured? Number of transactions? Simulation time? More important than the actual number of test cases or simulation time is the quality of the random tests. Issues such as input domain and oracle complexity define the

quality of the overall random test environment. Experience from previous projects can help find a decent answer to the question of the quantity of random tests.

The two figures below illustrates the difference between a good and a bad random test. The outermost quadrat represents all possible inputs, which in a complex system approaches infinity. The hexagon represents the input that is actually used in production. The ellipse is the total amount of bugs in the entire system and the circle symbolizes the coverage of the random test. *Figure 2* shows a situation where the random test hardly finds any bugs at all. *Figure 3* on the other hand is a utopia where almost all of the bugs in production are detected by the random test. It is important to understand that it is impossible to get the circle to cover all bugs. Random tests are always only a part of a larger test strategy. [6]



*Figure 2, Bad Random Test [6]*



*Figure 3, Good Random Test [6]*

#### 4.2.4 Trading simulations [5]

A market place system is very complex. Transaction sequences can virtually take an infinite number of forms and the system often supports the use of special order types such as stop-loss and combination orders. With these order types only one order can initiate a matching sequence including several hundred orders in different order books. A way to create appropriate trading sequences is to use trading simulators, also known as simulation actors.

Every simulation actor has a specific user profile in the trading environment. One actor can for example trade with hidden volume orders whereas another works with the change of order book states. Volume, size and other parameters of an order are randomized according to pre-defined conditions. A normal actor has about 10-30 different actions and each action has 2-3 randomized parameters.

<b>Actor</b>	<b>Description</b>
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 1: Examples of Trading Simulation Actors [6]*

When a simulation is set up a mix of different actors is used. The actors then execute their own actions according to the defined parameters independent of each other, creating a realistic trading environment. However it is important that the trading sequence can be repeated, this to allow debugging of an error by running the same sequence several times.

The major advantage with trading simulations is the ability to create complicated trading sequences. The problem however is how these simulations should be evaluated. Since the actors produce so complex sequences normal checks are generally not applicable.

However the evaluation of random trading simulations can be divided into three different categories:

- Trivial tests, for example constraint checks, are always applied when a simulation or test case is executed.
- Certain simpler types of explicit tests can be applied to the actors.
- The system under test does not crash.

The first category is actually very efficient. If an error occurs it will sooner or later break one of the general conditions checked by trivial tests. Furthermore, as the trading sequences get very complex just knowing that the system works properly is important information.

### **4.3 The Oracle**

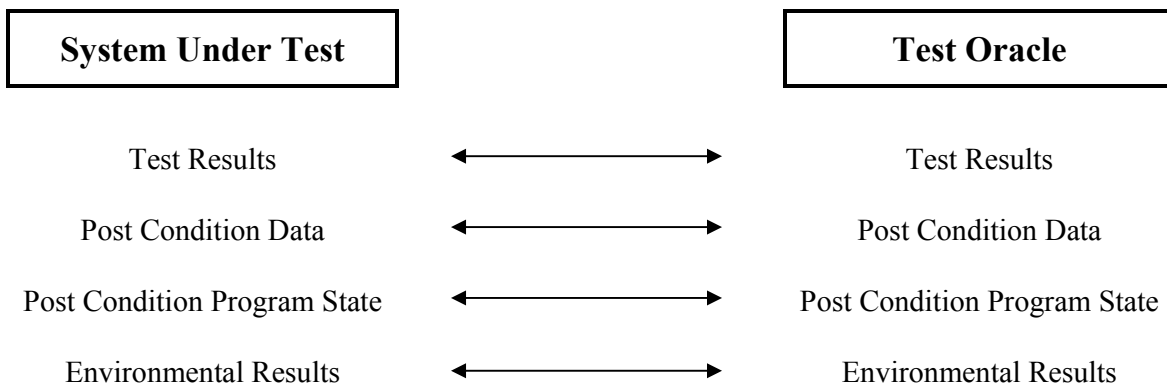
The key to successful testing is capture and comparison of results. The role of an oracle is to verify that the output from the system is correct according to the input (or at least reasonable). When designing an oracle there are a lot of factors to consider. Oracles can range from very simple programs that only monitor that the system runs properly to exact duplications of the system under test. A list of considerations is presented below, it gives an idea on the complexity of the process of designing a good oracle. [10]

- **Completeness of information**
  - Input coverage
  - Result coverage
  - Function coverage
  - Sufficiency
  - Types of errors possible
- **Accuracy of information**
  - How similar to system
    - Arithmetic accuracy
    - Statistically similar
  - How independent from system
    - Algorithms
    - Sub-programs and libraries
    - System platform
    - Operating environment
    - Close correspondence makes common mode faults more likely and reduces maintainability
  - How extensive
    - The more ways in which the oracle matches the system, for example the more complex the oracle, the more errors
  - Types of possible errors
    - Misses actual wrong value
    - Flags correct data as an error
    - Some oracles may allow one or even both types of errors
- **Usability**
  - Form of information
    - Bits and bytes
    - Electronic signals
    - Hardcopy and display
  - Location of information
  - Data set size
  - Fitness for intended use
  - Availability of comparators
  - Support in system environments
- **Maintainability**
  - COTS (Commercial Of The Shelf) or custom
    - Custom oracles can become more complex than the system itself
    - More complex oracles make more errors
  - Cost to keep correspondence through system changes
    - Test exercises
    - Test data
    - Tools
  - Additional support activities required
- **Complexity**
  - Correspondence with system
  - Coverage of system domains and functions
  - Accuracy of generated results
  - Maintenance cost to keep correspondence through system changes
    - Test exercises
    - Test data

- Test tool
      - Additional support activities required
- **Temporal relationships**
  - How fast to generate results
  - How fast to compare
  - When is the oracle run
  - When are results compared
- **Cost**
  - Creation or acquisition costs
  - Maintenance of oracle
  - Execution cost
  - Cost of comparison
  - Additional analysis of errors
  - Cost of misses
  - Cost of false alarms

There is a trade-off between cost and complexity when creating an oracle. The more complete it is the higher the development and maintenance cost will be. A true oracle can actually be more complex than the system that is tested. As the oracle gets more complicated the chance that a detected error is located in the oracle itself and not the system increases. There is also a chance that errors can be missed due to common mode errors, where the oracle and system produces the same wrong output. Furthermore a complex oracle is more vulnerable to changes in the operating environment. The maintenance cost can be very high if the conditions surrounding the test are constantly changing.

One must therefore understand that it is not enough to only compare the results from the oracle and system to verify correct behaviour. All conditions surrounding the test must be considered. For example the test environment, the current program state and so on. The picture below illustrates which circumstances have to be taken into account when setting up a random test with a verifying oracle.



*Fig 4, Test System vs. Oracle [12]*

Five different approaches are listed in [10] that explain some of the most used and successful ways to use oracles to test software, see *Table 2*. The oracle types are explained below with

emphasis on heuristic oracles. For random tests heuristic oracles are a good compromise between exact duplication of the system under test and no oracle at all.

	<b>No Oracle</b>	<b>True Oracle</b>	<b>Consistent Oracle</b>	<b>Self - Referential Oracle</b>	<b>Heuristic Oracle</b>
<b>Definition</b>	Does not check correctness of results.	Independent generation of all expected results.	Verifies current run results with a previous run (regression test).	Embedded answer within data.	Verifies some values as well as consistency of remaining values.
<b>Advantages</b>	Can run any amount of data.	No encountered errors go undetected.	Fastest method using an Oracle. Verification is straightforward and large amounts of data are easily generated.	Allows extensive post-test analysis. Verification is based on message contents. Can generate large amounts of complex data.	Faster and easier than True Oracle. Often less expensive to create and use.
<b>Disadvantages</b>	Only spectacular failures are detected. (e.g. crashes)	Expensive to implement. Complex and often time-consuming when run.	Original run may include undetected errors.	Must define answers and generate messages to contain them.	Can miss systematic errors.

*Table 2, Oracles [10]*

### 4.3.1 No Oracle

A cheap way of implementing automated random tests is to run them without any verification. This gets around the problem with false error reports and maintenance costs. This approach is inexpensive and tests can be run quickly. Of course the major disadvantage is that only spectacular errors such as crashes can be detected. [10]

### 4.3.2 True Oracle

A true oracle exactly reproduces the work of the system under test. They are both fed with the same input data and the outputs of the two systems are compared. True oracles are extremely expensive to develop and maintain, and are often used for subroutines in a larger programs but not complete systems. It is important that the oracle is implemented in a different way (for example another algorithm) than the system under test. This to ensure that the same design faults will not appear in both systems. The less the oracle has in common with the system the more confidence in the verification.

A true oracle does not have to be complete to be useful. It can be designed to work properly for a limited range of inputs. By choosing random input values from this domain functionality

for this input range can be tested. By using a random number generator with a known seed the sequence is repeatable and can be run again if a bug is found. [10]

#### **4.3.3 Consistent Oracle**

The consistent oracle compares the results from previous test runs with the current test. This is a normal method for regression tests, as it is an easy way to verify that no unwanted changes have occurred from one revision to another. For random tests the same seed is used, this reassures that the same input sequence will be selected for the different test runs. The approach does not tell if something is wrong but it exposes differences, differences that very well might be defects in the new version of the software. A drawback with this type of oracle is that historic faults will remain since wrong output will be identical in an old and a new test run. [10]

#### **4.3.4 Self-Referential Oracle**

For a self-referential oracle the correct results are built into the input data. When for example testing a data base system, one of the data fields could explain the expected relationship between fields or records. For random input the seed is included in the data set so reruns are possible. Test are often designed to create records with special characteristics, those characteristics are then included within the records themselves. [10]

#### **4.3.5 Heuristic Oracle**

A heuristic oracle uses simplified algorithms compared to the functions in the system to verify correctness. It is often possible to recognize a pattern in the output and this pattern can be used when designing the checks. The approach is especially useful when there is a nice and predictable relationship between the input and output. There is no guarantee that all errors are detected but it will tell if an output is reasonable or not, and that is often enough. In addition the number of false errors is expected to be low when using a heuristic oracle. [10]

The relationships that are identified must work for a range of input and output values. It is important that the heuristic holds for all the values in the defined range. An exception to this range makes it very difficult to design a good oracle and other approaches should be considered. [13]

For example, a heuristic strategy for a Swedish personal number might check that the value has ten digits, that the two digits that represent the month are not higher than twelve and so on. Additional constraint checks can be added and by narrowing down the valid area of the output the verification can be more and more exact. Sooner or later an error will lead to a conflict with a constraint check. For a market place system constraint checks can for example be: [6]

- No crossing prices (could be valid in some markets or situations)
- No negative volume of orders
- Usage of Shadow Order books (finds cancel of deleted orders etc.)

When finding the heuristic there are a few rules of thumb that can be used. There are almost always simpler patterns behind complex algorithms. Sometimes just considering a subset of the input variables show a pattern that was not visible at start. If several relationships can be

found it is often wise to choose the simplest one. A simple heuristic means an uncomplicated, reliable and fast oracle. The complexity of the oracle can be improved later on if it turns out that it will not find the required number of errors. [13]

The system under test can often be split into segments. These segments can then be analyzed independently, and different heuristics can be used to verify the separate outputs. Results can also often be divided into things we know to expect, and what is impossible to predict. The known results do not have to be verified by the oracle and they might also be valuable when finding the heuristic for the unknown results. When the input consists of a sequence rather than a single value the starting value and the number of values can be a simplified representation of the whole sequence. If the variation of the values in a sequence is important fixed increments can be used to calculate all values by only knowing the first one and the increment. [13]

When choosing heuristic it is important to consider which failures will go undetected and which the oracle will find. A decision must then be taken if one is willing to accept that risk. There is no end in itself to build an extremely complex oracle that can detect every single error that might occur. A trade-off between functionality and cost must be made.

If no pattern is recognized in the data to be analyzed heuristic oracles will not apply. If the pattern is too complex there is a large chance that the oracle gets complex as well. This increases the probability that errors are introduced in the design of the heuristics. The task of finding a nice pattern is crucial to the development of a good heuristic oracle.

## **4.4 The Output**

The output of a random test cannot only be used to verify correct behaviour (with the help of an oracle) of a complex system. The output will also reveal if the wanted coverage of the tests was obtained. Did the expected transactions in a market place system actually occur? Another way of using the output is to measure the stability of the system. The problem of estimating the reliability of a software system will be thoroughly discussed.

### **4.4.1 Reliability**

Engineers working with software reliability issues are divided on how to approach the problem with systematic faults. Some argue that since a fault will occur every time a certain part of the code is executed it cannot be seen as random. Therefore statistical analysis is impossible to use to measure the reliability of the system. Others say that in a complex software system a fault can take an almost infinite number of forms. This means that unless an error is already detected it can be seen as random, and statistical analysis applies. Engineers sceptical to statistical analysis on computer software have not come up with an alternative method to deal with the problem. The practice to see software faults as random has also proved to work well in several projects and most people now believe that it has a place in the analysis of software reliability. [7]

A market place system requires a very high reliability. The problem with systems that have to be ultra reliable is that they might need impractically long hours of tests to assure adequate reliability. But there are ways to partially get around this issue and ensure reliability with a reasonable number of test hours. Only a limited number of functions that can make the entire

system crash have to be extremely reliable. Secondly, execution time and clock time are not the same. The time of execution is only a small fraction of the total time and by running critical operations often, total test hours can be reduced. At last, because processing time is getting cheaper and cheaper it is possible to design parallel test environments that work hundreds of times faster than a serial one. Testing of extremely reliable software is demanding, but possible. [14]

#### 4.4.2 Reliability models

When creating reliability models there are a number of general assumptions that often are made. Some of the more common ones are listed below: [15]

- *Times between failure are constant*

This assumption is made in all “time to failure”-models. For randomly chosen test input it is valid, but often tests will be concentrated on fault-intense parts of the code.

- *No new faults are introduced during the fault removal process*

This is not always true. If faults are removed immediately after detection the removal might cause another error somewhere else in the code.

- *Failure rate is proportional to the number of remaining faults*

This implies that all remaining faults have the same probability of being detected. But the case is often that some parts of the code are executed more frequently than others and hence error detection is less likely in the rarely executed code.

- *Reliability is a function of the number of remaining faults*

Similar to the previous point. This assumes that all remaining faults are likely to appear during execution of the code. If some areas of the code are executed more often than others the assumption will not hold.

- *Failure rate increases between failures*

This assumption implies that the chance of finding a fault increases when test time increases in a given failure interval. This is true when testing products that wear out by time, which is not the case with software.

There are several ways to create these models that approximate reliability of a software system:

- One approach is to consider the failure rate ( $\lambda$ ) of a system. The failure rate is simply the number of expected failures per hour. For example, a system that fails once in 1000 hours of operation has a failure rate of 1/1000. To simplify calculations the failure rate is often seen as time-invariant. This means that the failure rate is constant over time. It is also possible to calculate with time-variant failure rates and this is often modelled by the Weibull distribution. The rest of this text will however assume constant failure rates. [7]

- If the input is random and the system handles large amounts of data the distribution can be seen as binomial. This can be modelled by Poisson approximation as probability of failure per time unit. The distribution of the output is exponential. [15] By using this value it is possible to continue and estimate the mean time to failure (MTTF). If the mean time to failure is compared to the failure rate it is found that they are simply the inverse of each other. ( $MTTF = 1/\lambda$ ) [7]

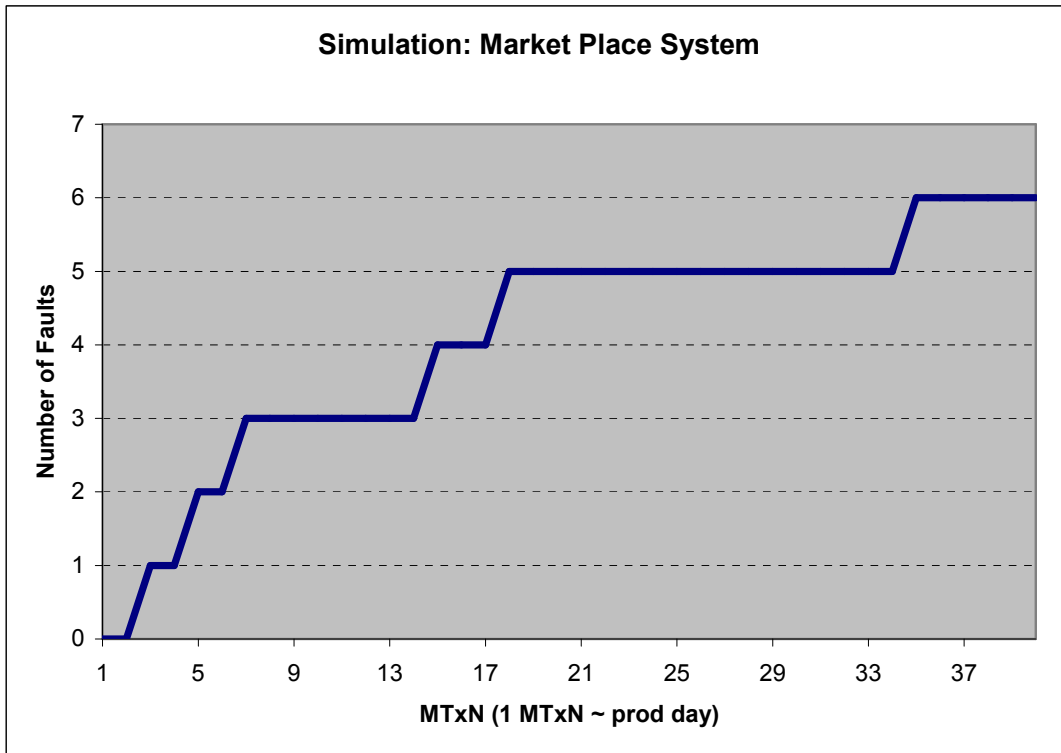


Fig 5, Simulation (Faults/transactions) [6]

From the graph it is possible to make a rough estimate of the MTTF. In the first 20 MTxN (millions of transactions) five faults are found, which states a MTTF of approximately 4 MTxN (4 prod days). In the later stage of the graph (20 MTxN -> 40 MTxN) when detected faults have been removed only one error is detected and hence the MTTF has increased to around 20 production days.

- Fault count modules are used during the debugging phase, and are used to model the number of errors in a specific test interval. These intervals are defined before the test starts and the monitored numbers of failures are treated as a random variable. When detected faults are corrected the number of found errors is expected to decrease. The idea behind most reliability models is to use a Poisson distribution whose parameters change according to the model. The Poisson distribution is often a great model when dealing with applications where there is an interest in the number of occurrences. [15]
- The fault-seeding model introduces faults into the program under test. The number of detected faults can be compared to the faults injected, and an estimate of the total number of faults can be calculated. [15]

- It is also possible to use Markov models to approximate reliability. It uses states and state transitions to model system behaviour. The systems total reliability is calculated from estimations of smaller individual modules. [15]
- Depending on which programming language is used the number of faults per line of executable code can be seen as constant. This model says nothing about the severity of the errors and do not take into account things such as the skill of the programmer etcetera. Therefore the model is not considered very reliable but can be used to get a first estimation of the number of faults in the system. [15]

Besides finding the faults in the software an estimation of the consequence of the faults has to be made. As it is impossible to remove all defects, the reliability of the system will be determined by how often faulty code is executed and the effects of this execution. Some errors may have virtually no impact on the system, whereas others are catastrophic. [7]

#### **4.4.3 Reliability assessment**

When a system has been created it is important to show that it meets the specified reliability requirements. The assumptions made during development must be confirmed by testing. The most straight forward way of testing the reliability would of course be to run the system long enough to prove that it meets its requirements. This is often practical impossible since most system have high demands on the mean time to failure. The system might have to be run for years to get a fair estimate of the reliability. [7]

An alternative way is to construct a large number of identical systems that are executed simultaneously and then the average reliability values for the entire group are determined. For small low-cost applications that do not need extremely high reliability this is often a good method, especially if accelerated life testing can be used to reduce the testing time. However, for a complex software system that requires high reliability it is rarely a good idea. The system is often to expensive to produce in large numbers, and even if mass-production is possible testing times will still be to long to make it practical possible. [7]

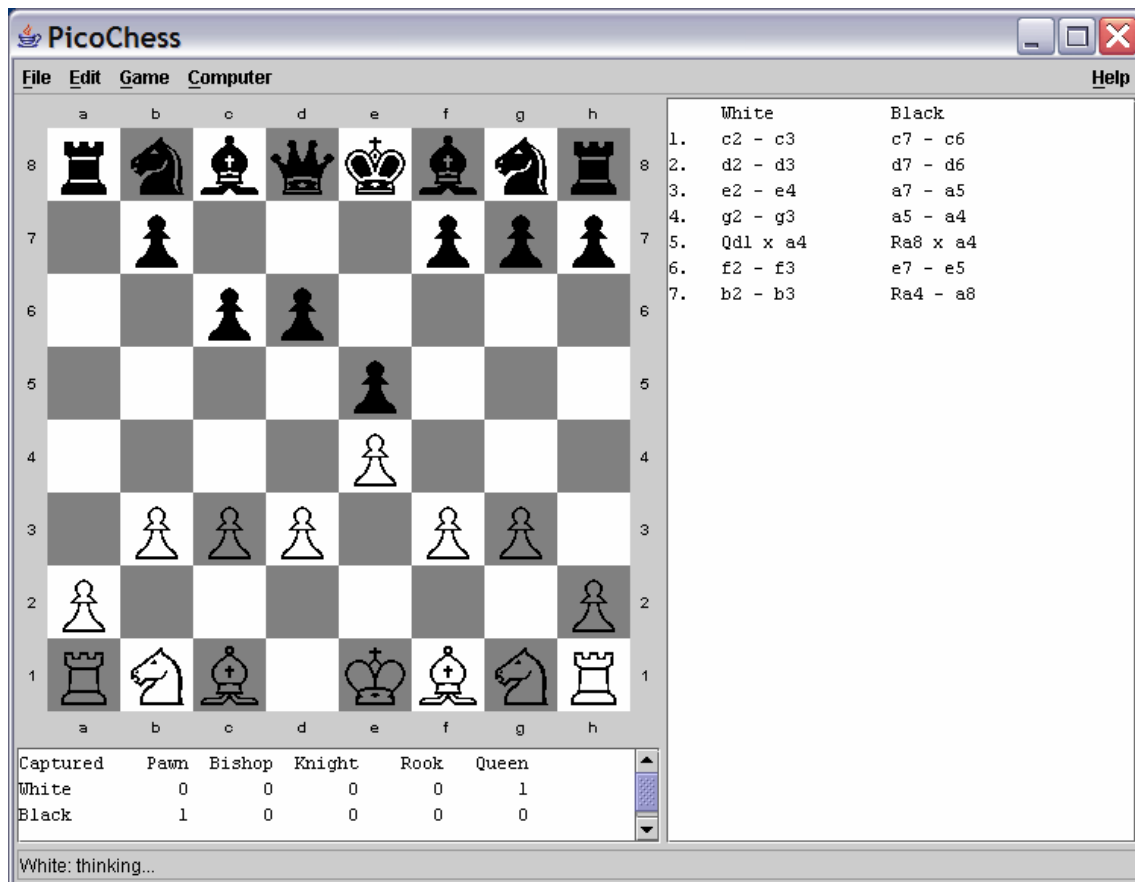
A well known problem with systems that require very long mean time to failure is that it generally is impossible to prove that the software meet these requirements. At present most critical computer systems demand a reliability that is several orders of magnitude better than can be demonstrated by testing. Therefore acceptance and certification of such systems has to be made using reliability models and appropriate development techniques. [7]

## 5 Implementation and Results

A trading system is large and very complex. Instead of working toward a real system a substitute program was chosen that has the same characteristics as a trading system, although smaller and less complex. The system must allow sequences of operations to be performed in a similar way that a market place system works. Research and discussion with people at OMX Technology showed that a chess simulator could be a good trade off between the complexity of a trading system and the easy of use of a smaller program. A chess game consists of sequences of moves in the same way that a market place consists of transaction sequences. These moves can build complex scenarios in the same way as multiple transactions do in a market system.

### 5.1 Picochess

Picochess is a small chess simulator with a simple graphical interface that allows two player to play against each other or a single player to play against an artificial intelligence that is based on a well known algorithm. The program was developed by Daniel Beer and is free to distribute and modify under the GNU General Public License. Picochess is written in the object oriented programming language Java.



*Figure 6, Picochess GUI*

## 5.2 Modifications

Picochess was modified allowing two artificial intelligences to play against each other without human intervention. This is similar to the simulations in a market place system where actors have different roles and interact with each other. To avoid that all games look alike according to the moves chosen by the artificial intelligence a randomizer was introduced that made a random move a predefined percentage of the times a player was to act.

A new Java class called Error was introduced to the Picochess program. The class contains functions that will trigger a flag to be set if a certain pattern is recognized in the game between the two artificial intelligences. The exact pattern itself is not as central as the importance that it is a situation that can occur in a chess game with a certain probability that is larger than zero and smaller than one.

The following patterns were chosen and implemented in the Error class of the Picochess program:

- A      A knight captures a knight
- B      A pawn is moved three times in a row by the same player
- C      All kings and queens are located on black squares
- D      The queens are located in squares adjacent to each other
- E      A player checks two or more times in a row

When these conditions are combined there are 32 ( $2^5$ ) different combinations that can be the result of one chess game. For example only conditions A and C happened during the game giving the result AC, or all conditions were met giving the result ABCDE. These combinations can represent virtual faults of various severity in the system. For example can the combination ABCDE be treated as a show stopper which causes the complete program to crash.

## 5.3 Simulations

100 simulation games were run and the result of each game were recorded. A larger number of simulations are desirable but with every simulation taking approximately 30 minutes to one hour time was the limiting factor. Although 100 simulations is a large enough sample space to get an idea on the characteristics of the output data.

Every simulation resulted in combination of the letter A, B, C, D and E mirroring the five different events that are listed earlier.

## 5.4 Analysis

When analyzing the results there are some things to remember. Software has no aging property. When a software fault is removed, it will never cause the same failure again.

Measuring fault intensity in a software program is useful to answer some of the following questions:

- Does the software meet the desired quality level?
- Does the release have more faults than expected?
- How does the new release compare to previous ones?

The following figures show the number of faults found during 100 simulations. *Figure 7* shows all faults independent of each other. If faults A and C take place the values for A, C and AC are increased by one. *Figure 8* on the other hand only show the exact fault combination, this means that only the value AC is increased if both fault A and C occur.

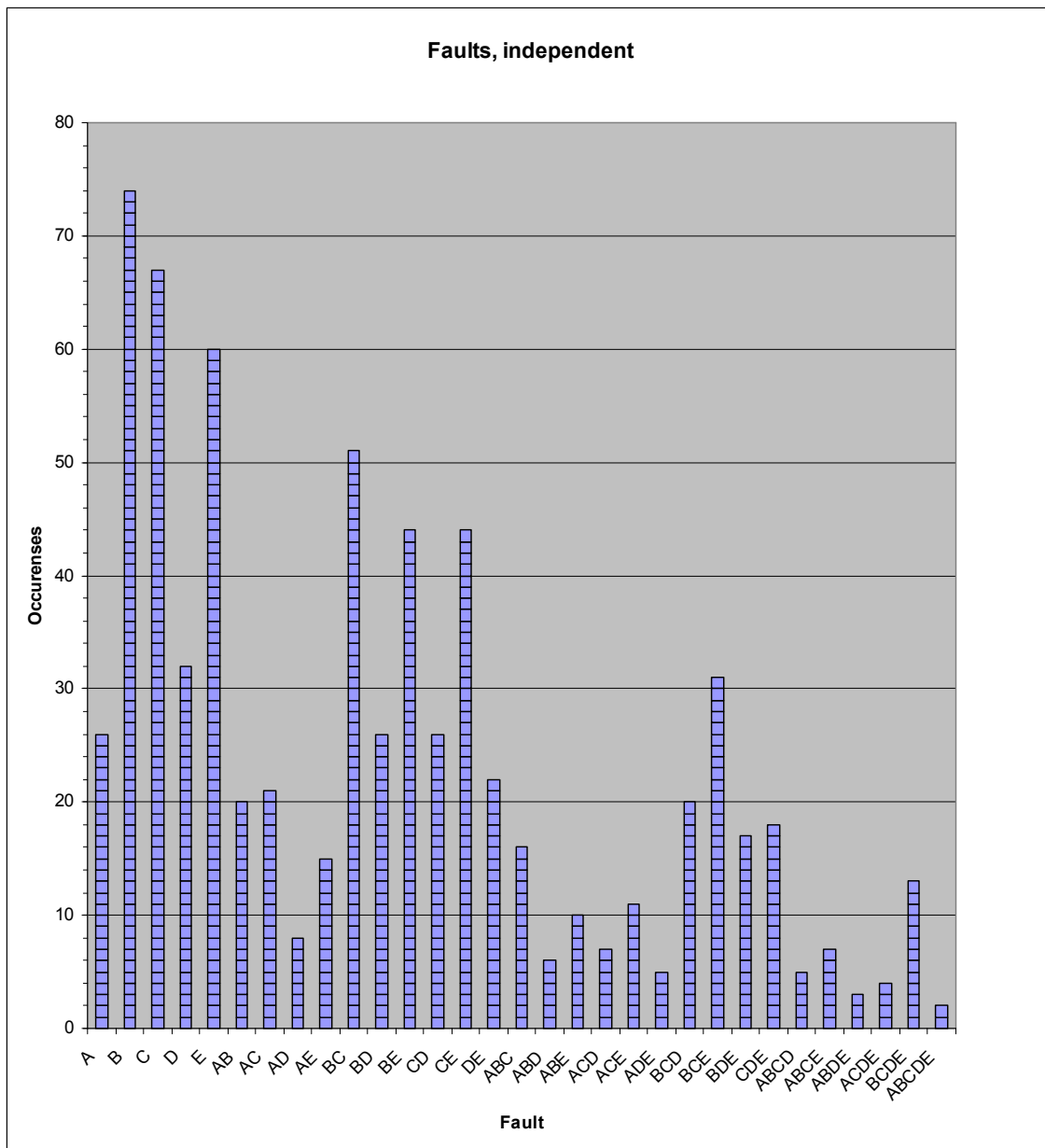


Fig 7, Faults, independent

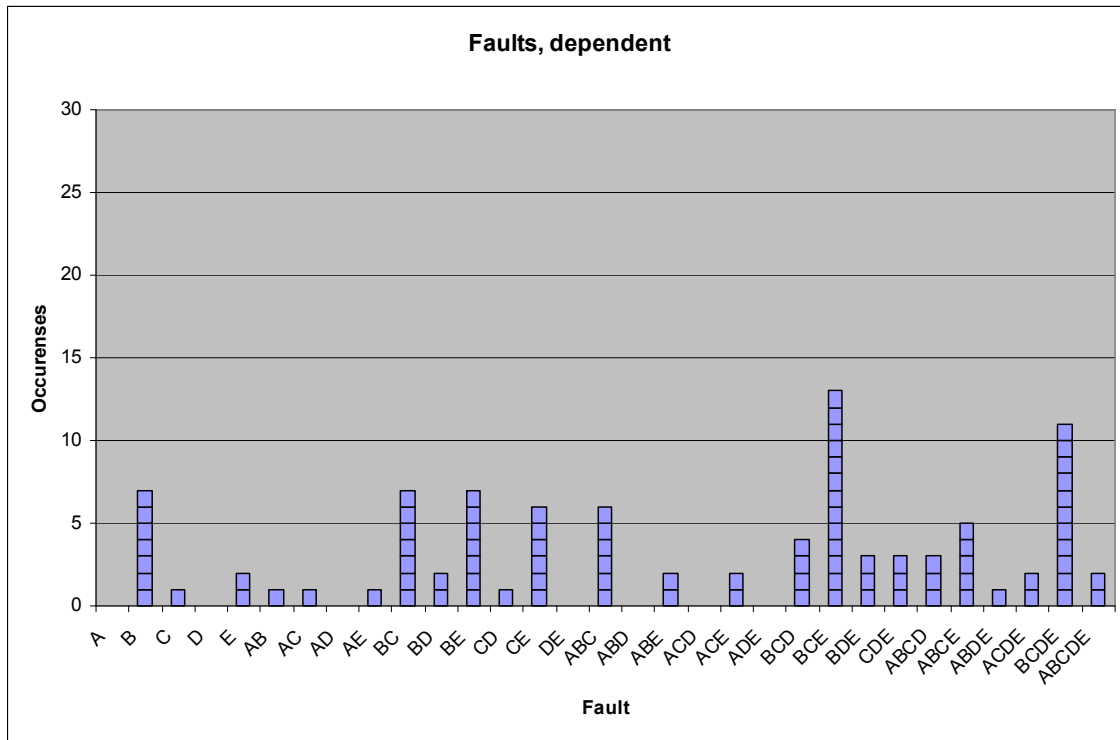


Fig 8, Faults, dependent

These charts give an idea of the frequency of the different faults and combinations of faults injected in the system. From *figure 7* we can see that all 32 combinations of faults occur at some time. If all faults could be exposed individually there is a great chance that the vast majority of all error in the system has been found. The ratio between the number of injected faults and found faults is one in this case. We can therefore, in this simplified example, expect the ratio between faults found by test cases and the number of real faults in the system to be close to one as well.

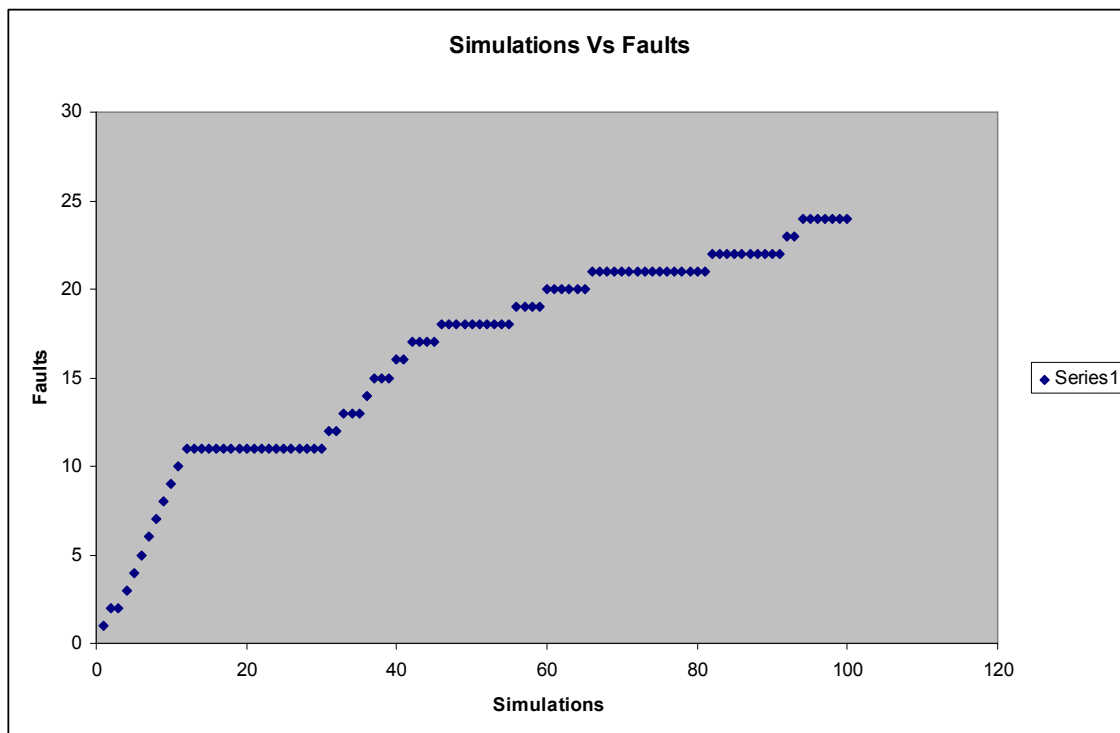
A simulation sample such as the one in *figure 7* gives an estimate of the frequency of all faults. It is easy to realize that faults B, C and E are overrepresented compared to faults A and D. Hence all fault combinations containing the high frequency faults are more common than the ones without. This information can be used when reviewing a number of real faults as well. In a situation where there is a tight deadline and all faults are considered equally important this simulation shows which faults should be dealt with first. By removing faults B, C and E the reliability of the system increases a great deal since all combinations only containing these three faults will also be removed.

*Figure 8* provides a different view of the number of faults. Here a fault is only recorded if it meets the exact combination of detected and undetected faults. Hence the combination ABC requires that faults A, B and C are detected and that faults D and E are undetected. This is a more realistic scenario in a market place system where certain combinations of transactions can cause the system to crash, although the transactions themselves are not dangerous.

In *figure 8*, 25 out of the total 32 faults are detected, which is approximately 78 %. A rough estimate of the faults missed by regular test cases would in this case be 22 %. For example, if the regular tests have found 55 faults we can expect the system to have a total of 70 faults.

The ratio between the faults found in *figure 8* compared to the ones found in *figure 7* is approximately 1:8. This gives a good understanding that as the faults become more complex and requires exact combinations of transactions to be triggered, they become very difficult to find. This simple example only includes five different faults and combinations of them, as the system grows to the size of a complex market place system the combinations of different transactions that can trigger faults are practically infinite.

From the simulations it is possible to look at how many faults have been found after a specific number of simulations. This curve normally flattens out as the number of simulations increase since every fault only is counted once. After a large enough number of samples it is possible to show if the number of faults found per simulation is within the predefined requirements.



*Fig 9, Simulation vs. Faults*

*Figure 9* shows that a large numbers of faults are found in the earlier simulations. To get an even better estimate of when the absolute majority of the faults have been found the sample space would have to be larger.

The Simulation vs. Faults graph also gives an opportunity to get an estimate of the Mean Time To Failure. During the first 50 simulations 18 faults are found which gives a MTTF of about 2.8 simulations, the system fails about every third time it is run. When these faults have been removed only 7 faults are found during the last 50 simulations. This represents a MTTF

of 7.1 simulations. As the number of simulations increases a more reliable approximation of the true MTTF is obtained.

# 6 Discussion

---

## 6.1 Issues

### 6.1.1 Platform

The practical part of the master thesis was first intended to be done on a real market place system. However the complexity of the system as well as the problem with configuration and support made this impossible. A stand alone system that allowed the author to do major changes without causing problem for test engineers currently working on the system was difficult to achieve. A compromise with a smaller program had to be made.

### 6.1.2 Simulations

A higher number of simulated chess games would be preferable to get a larger sample space. The problem with the long simulation time was the artificial intelligence built into the chess program. Every move would take up to one minute if the algorithm was to run until it found what it considered to be the best move.

It was possible to force the algorithm to stop after a predefined number of seconds and chose the best move found so far. This approach however caused the chess game to be a game of randomly moved chess pieces, and almost always resulting in a draw with only the two kings remaining on the board. This is not a realistic scenario and the fault combinations found with this approach would be irrelevant. When allowing the algorithm to run until it finished games could end in only 10-20 moves as well as go all the way with capturing of almost every piece on the board except the kings.

## 6.2 Further Work

Software reliability is a popular research subject and a number of models have been introduced in the last decades. When using fault injection there are models more advanced than just comparing the ratio between found and undetected inserted faults. One example is Mill's Hypergeometric Model. It uses a joint probability function to get an estimate of the number of indigenous faults in the system.

There are a lot of software reliability models based on other inputs than fault seeding that would be interesting to further look at. A short description of the different types can be found in [16].

## 6.3 Summary

Random tests in combination with regular tests can be a powerful tool to ensure high reliability in safety critical computer software systems. Faults that are not found by the normal test portfolio can be triggered when a random set of input data is used. Complex combinations of transactions that cause the system to fail can also be found and dealt with.

This thesis gives a description of a trading system developed by OMX Technology and how this system is tested today. It provides an overview of random test and the benefits of these tests. Different input and output domains are discussed and various oracle approaches are explained.

The small practical study that used the fault injection technique shows how to get an estimate of the systems reliability. Information about the frequency of different faults gives a hint of what parts of the code are more likely to contain high numbers of undetected faults. The simulations also give the ability to obtain an estimate of the Mean Time To Failure for the tested system, although the sample space was too small to reach any certain conclusions in this case.

Overall random tests are a great way to further improve the quality of testing when working with high reliability computer software programs.

# References

- [1] **[LINDHOLM, WESTERGREN]** *Informationshantering i börsnoterade företag*, Ekonomiska Institutionen Linköpings Universitet, 2003
- [2] **[OMX]** *SAXESS Marketplace solution*, [http://www.omxgroup.com/pdf/SAXESS\\_8%20sidig.pdf](http://www.omxgroup.com/pdf/SAXESS_8%20sidig.pdf)
- [3] **[OMX]** *SAXESS Software Product Description*, v 40.19, 2004
- [4] **[OMX]** *SAXESS Manual*, 2004
- [5] **[SELLBERG]** *MEXUS – main design ideas*, v 1.3, 2003, <http://www.stickyminds.com/getfile.asp?ot=XML&id=6775&fn=XDD6775filelistfilename1%2Edoc>
- [6] **[WAHLBERG]** *OM HEX Test Strategy Handbook, Chapter: Random Tests*, 2004
- [7] **[STOREY]** *Safety-Critical Computer Systems*, Prentice Hall, ISBN 0-201-42787-7, 1996
- [8] **[HOFFMAN]** *Mutating Automated Tests*, Star East 2000, 2000, <http://www.stickyminds.com/getfile.asp?ot=XML&id=2051&fn=XDD2051filelistfilename1%2Edoc>
- [9] **[KANER]** *Architectures of Test Automation*, 2000, <http://www.kaner.com/testarch.html>
- [10] **[HOFFMAN]** *Using Oracles In Test Automation*, 2001, <http://www.stickyminds.com/getfile.asp?ot=XML&id=2969&fn=XUS801608file1%2EPDF>
- [11] **[WHITTAKER]** *Random Testing and “App Compat” – The ten commandments of Software Testing*, 1996, <http://www.stickyminds.com/sitewide.asp?ObjectId=6392&Function=DETAILBROWSE&ObjectType=COL>
- [12] **[KANER]** *High Volume Test Automation*, Star East, 2004, <http://www.kaner.com/pdfs/highvolCSTER.pdf>
- [13] **[HOFFMAN]** *Heuristic Test Oracles*, 1999, <http://www.softwarequalitymethods.com/SQM/Papers/HeuristicPaper.pdf>
- [14] **[HOFFMAN]** *Mutating Automated Tests*, Star East 2000, 2000, <http://www.softwarequalitymethods.com/SQM/Slides/MutatingAutoTestsSlides.pdf>
- [15] **[JOHANSSON]** *Safety-Critical Control System. The Challenge of Migrating from Hardware to Software*, 1996
- [16] **[GOKHALE, MARINOS, TRIVEDI]**, *Important Milestones in Software Reliability Modelling*, 1996, [http://www.ee.duke.edu/~ssg/papers/journal\\_sae.ps](http://www.ee.duke.edu/~ssg/papers/journal_sae.ps)
- [17] **[OMX]** *OMX Technology website*, [http://www.omxgroup.com/solutions/en/omxtech\\_about\\_omx\\_technology.aspx](http://www.omxgroup.com/solutions/en/omxtech_about_omx_technology.aspx)