Automated Debugging in a Trading System

Author
Saeed Ansari Ramandi

Master's thesis. Stockholm, December 2011
School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology

Examiner
Christian Schulte

Supervisors

Noah Höjeberg

Roberto Castaneda

Lars Wahlberg

Abstract

Verifying the reliability and functionality of a complex system like a trading system is highly demanding since failure in such a system can cause serious economic problems. Automated random testing is a good solution to find new and rare failures in such a system. Test cases in random testing usually contain a long sequence of actions that debugging them manually to find the root cause of the failure is a very boring and tiresome task.

This thesis aims to create a model for automating the task of the debugging to reduce the failed test case to an equivalent test case that only contains relevant actions that together cause the failure. Delta debugging is the core algorithm of the model that simplifies a failed test case by successive testing.

The target of the project is TRADExpress system of Cinnober Financial Technology AB. The model is integrated to the random testing framework of the TRADExpress system.

Keywords: Delta debugging, automated debugging, trading system, random testing

Contents

List of Figures

List of Tables

1 Introduction

Debugging computer programs, the process of identifying and correcting the root cause of a failure is a difficult, labor-intensive and time consuming activity in software development. When one observes a failure, setups a number of hypotheses to track the effects and causes of the failure which one confirms or rejects. Frequent making and refuting hypotheses is very tedious and chaotic; the solution to this problem is systematic and automated debugging that makes the bug fixing faster and efficient and more effectively increases productivity [4].

By automating the debugging one can take the advantages of [3]:

- Reusing the existing tests (for instance, to test a new version of a program)
- Performing tests which are difficult or impossible to carry out manually (such as massive random tests)
- Making tests repeatable
- Increasing confidence in the software

The Delta Debugging algorithm is an automated and systematic approach that simplifies a failing test case by narrowing down failure-inducing circumstances by successive testing till remaining a minimal set that still produces the failure. The Delta Debugging algorithm will be fully covered in detail in chapter 4.

1.1 Previous works at Cinnober

So far several master theses were developed in Cinnober about random testing. In [19], the author conducts a theoretical study of random tests and develops a random test framework using trading simulations in order to find low frequent errors in the trading system. The tests are analyzed by various types of test oracles and at last the best oracle that fits in random testing is selected. This work is extended in [12] by analyzing the input data and providing realistic input domain for the random testing. Both works [19] and [12] lack an automated system to verify the correctness of the trading output. This problem is solved in [6] where constraint programming is employed to develop a test oracle that automatically predicts the system's output correctness.

1.2 Problem statement

Randomized unit test cases are very effective in finding low frequency bugs in a program. However, failed test cases often comprise long sequences of method calls; reproducing and manually debugging them to find the failure root is cumbersome. In Cinnober's TRADExpress system, we are interested in sequence of transactions that together make the failure; usually there are massive numbers of transactions that are irrelevant to the failure and reproducing those takes a lot of time. Furthermore, state dependency among the transactions makes the debugging

process even more difficult. The need of designing a model for this system to make the debugging task more efficient, faster and easier is undeniable.

1.3 Goals

The goal of this project is to design and develop a model for randomized test case simplification. The model will be integrated into existing random testing framework of Cinnober TRADExpress system. The work requires producing transactions' log for the purpose of replaying subsets of a given random test. The failing test case should be minimized to reach the minimal failing test case which contains only the transactions that are relevant to the failure. For evaluation purpose, the model is fed with artificial failures to measure the time complexity in several situations of interest.

1.4 Related Works

Delta debugging as a general algorithm for minimizing the failure-inducing inputs is utilized in many other researches. These works have focused either to apply delta debugging on inputs that are difficult to debug or to speed up the algorithm.

In [1], delta debugging algorithm is used to isolate interactions that are relevant for failure in a large system. In [6] and [18], this algorithm is applied on chain of system variables and threads respectively.

In [2], the delta debugging is used in combination with static slicing approach to make the test case minimization process faster. This method is applicable to program statements since it finds the slice of the program that is relevant to the failure. In another research [16], authors proposed a method called Hierarchical Delta Debugging to speed up the delta debugging and increase its output quality on tree structured inputs.

In this thesis work, the delta debugging will be applied to the transactions communicated in a trading system to isolate the chain of the transactions that cause a failure.

2 Random Testing

Software development is always subject to the errors defined as human action that produces an incorrect result. Errors or mistakes can occur in design or build phases of software development. These mistakes are known as defects or bugs. Any defect that may cause the system to fail after system execution is called failure. The goal of software testing is increasing the quality of the software by minimizing the number of defects and also to check if the software meets the system requirements and as well as user and customer's needs and expectations [13].

Software testing comprises two types of tests: manual software testing and automated software testing. The former is performed by a human that tests different combination of inputs and compares the result with expected result. The latter can be performed by a tool that tests pre-defined actions and compares the output with the expected results and reports failure or success of the test [21].

A more advanced level in test automation includes auto-generation of test cases, and mechanisms to automatically detect system failures called Random testing. In random testing (Monte-Carlo testing) the program under test is fed with randomly generated inputs from a preselected input domain. The correctness of the output is determined by an oracle that analyzes the expected output with the real output of the program [19].

The three different challenges in designing a successful random test environment are [17]: the input, the oracle and the output.

2.1 The Input

The input is determined by three characteristics; the domain from which input data is extracted, the type of distribution that the test follows either realistic or unrealistic and the size of the input. Good design of these characteristics is a key success to a reliable result achievement [20].

2.1.1 Domain

The input domain of a complex system is practically infinite but it is possible to partition the input into sub domains and feed the system with the data of the sub domains. Employing this strategy makes it possible to test functionality of the different parts of the program [17].

2.1.2 Distribution

The input can be generated from realistic and unrealistic distributions; to achieve a successful random test strategy both distributions are important. A trivial way of obtaining realistic data is using the historical data of the previous trading days which tests the existing functionality hence helps to assess the reliability of the system. By feeding the system with unrealistic data it is

possible to generate sequence of transactions that have not been executed in the system to find the low frequent bugs in the system [17].

2.1.3 Size

The number of actions in a test case to be generated and the running time in random testing depends on the purpose of the test and the structure of the system. If the intention of the test is to test the reliability of the program, the test should generate the realistic data in enough time.

2.2 Oracles

An oracle is a mechanism to verify if the output of the system under test is the one expected based on the input. Oracles have different complexity with regard to the goal of the testing and type of the software. A software program based on the multiplicity of input forms or different characteristics in a program (for example a program's results may include computed functions, screen navigations, and synchronous event handling) may require several oracles [15]. The basic structure of random testing with an oracle is shown in figure 2.1.

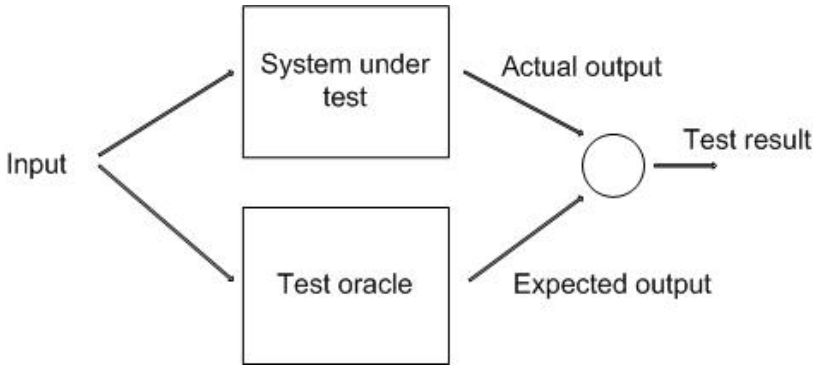

Figure 2.1: Random testing with an oracle

2.2.1 Oracle Characteristics

There are many tradeoffs in design of a test oracle that must be taken into account. The more complete an oracle is the more complex it has to be. An oracle with better prediction about program state and environment conditions is more dependent to the software under test (SUT). This dependency makes the oracle maintenance more difficult and also faults may be missed as both SUT and the oracle may contain the fault [15].

2.2.2 Types of Oracles

Oracles vary widely in their characteristics; the types of the oracles can be categorized based upon the outputs from the oracle rather than the method of generation of the results. The oracles are identified in four types: True, Heuristic, Consistent and Self-referential [15].

True Oracle
This type of oracle has no dependency to the SUT and is implemented using independent platform, algorithms, code and etc. The SUT and the oracle are fed with the same data for results comparison. True oracles are extremely expensive to develop and maintain, and are often used to test the system for specific input sub domains.

Heuristic Oracle
A heuristic oracle uses simplified algorithms and the verification often is done by checking the output based on the detected pattern in it. It cannot exactly verify that the result is correct, but can assess whether the result is likely to be correct or incorrect. The heuristic oracle is very easy to implement compared to true oracles and runs much faster and will find most faults. On the other hand, historical defects are likely to remain undetected, as the potential wrong output does not change over the different runs.

Consistent Oracle
A consistent oracle uses the results from one test run as the expected output for the next run. This oracle is useful for regression test to verify that changes do not affect the program correctness. The defect of this oracle is that the historical faults remain since wrong output will be identical in an old and a new test.

Self-referential Oracle
In a self-referential oracle, the results are embedded in the input as part of the test mechanism. When for example testing a database system, one of the data fields could explain the expected relationship between fields or records. A random number seed can be included in the input data so the test can be rerun with the same random number series. 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 [15].

2.3 The output

The output of a random test not only reveals the defects but also shows if the system is reliable. Software reliability is defined as "the probability of failure-free software operation for a specified period of time in a specified environment" [14]. In this section, fundamentals of software reliability are covered in some details.

2.3.1 Reliability models

Software reliability models are divided into two subcategories: prediction modeling and estimation modeling. Both models are based on failure data observation and the statistical data analysis. Prediction models use the historical data and predict the reliability before system deployment whereas estimation models use data from the current software development and estimate the number of defects has remained in the system or the time between failures [19].

Reliability models require some assumptions that some of the common ones are listed below [20]:

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

2.3.2 Reliability estimation

One of the ways to measure the reliability of a system is measuring the rate of the failure (λ). By measuring the time between each failure we can estimate the Mean Time Between Failures (MTBF) and thus failure rate λ, where λ = 1 / MTBF. The number of failures in a period of time is modeled as a random variable X ϵ Poison (λ). The probability of k faults occurring in the time T is calculated as:

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

Formula 2.1

3 The Trading System

A financial market is a system where sellers and buyers exchange many different physical or non-physical items. By introducing electronic trading, traditional financial markets have extremely changed during last decades. Now, majority of financial markets utilize computer programs to make the order movements faster by excluding the direct human intervention and automating the trade's transactions [7].

The TRADExpress developed by Cinnober Company is the target system of this master thesis. It is developed in Java and its robust infrastructure enables high-content, customized solutions to be delivered with short time-to-market [11].

3.1 Structure

This section gives an overview of the main structure of markets.

3.1.1 Type of the market

Trading in markets comprises exchanging of physical or non-physical valuable items, so called, financial instruments (or instruments). Some markets are specialized for exchanging in specific types of instruments whereas TRADExpress handles variety of instrument types within the same solution such as equities, commodity futures, warrants and etc [7].

3.1.2 Market participants

Market participants can be defined as the people and institutions that interact in a financial market. The set of actors varies in a market based on the market type. However, it is possible to identify four main participants on a financial market place [7]:

- Customers: invest/disinvest their money by means of buying or selling securities. Customers could be institutions or physical persons.
- Brokers: Act as an agent on behalf of customers and facilitate the trade for the customers by providing the client with information and advices.
- Dealers: trade on own behalf, assuming thus the risk of their operations
- Market makers: Or specialists have a pivotal position in market places and make their money selling to a higher price than they buy to. Add continuity to the market trade flow by ensuring that there is always a counterpart to trade with.

3.2 Architecture

One of the main concerns in TRADExpress design is customization which differentiates it from other trading systems. To fulfill this requirement the products are formed of three abstraction layers (Figure 3.1):

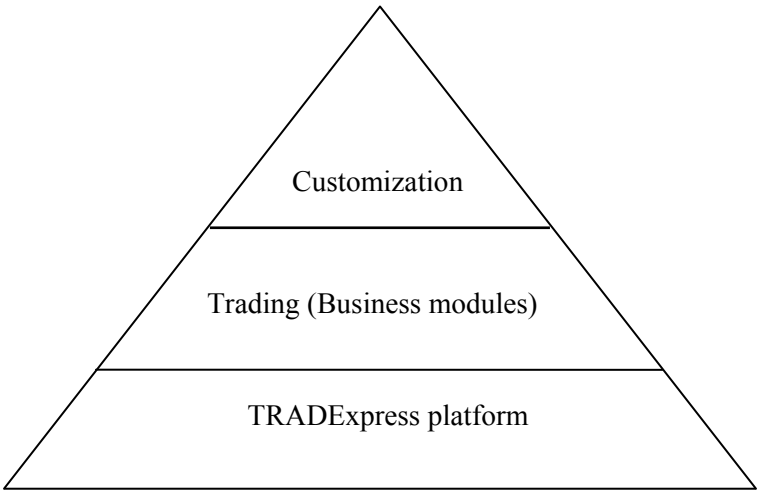

Figure 3.1: TRADExpress System Architecture

- Customization and Configuration layer: allows adding customer specific business rules, trading strategies, etc.
- Trading layer: includes business component library and application components
- Platform layer: includes middle ware, persistence and application infrastructure

The layered architecture of the servers facilitates developers with fast customization and furthermore provides extensibility, scalability, reliability, flexibility and low latency [20][21].

TRADExpress trading system is made up several servers that can communicate with the clients via two protocols: External Messaging Application Programming Interface (EMAPI) and Financial Information eXchange (FIX). The EMAPI is a session based API that handles lower level transport of messages using TCP/IP and multicast. FIX is a standard, widely used protocol for trading systems maintained by FIX Protocol Ltd [8][20].

Figure 3.2 presents the servers that provide the basic functionality of TRADExpress:

- Trading Application multipleXer (TAX): handles connections to external systems, converts external protocols and routes transactions
- Matching Engine (ME): handles the business functionality of the system: processing of trade orders, information dissemination to different connections, etc.
- Query Server (QS): maintains a copy of the state for different instruments, and serves some client queries in order to lower the load of the ME.
- Common Data (CD): The repository for all reference data and broadcasts reference data changes

- History Server (HS): stores historic events in a database (such as order events and trades) and serves queries from TAX

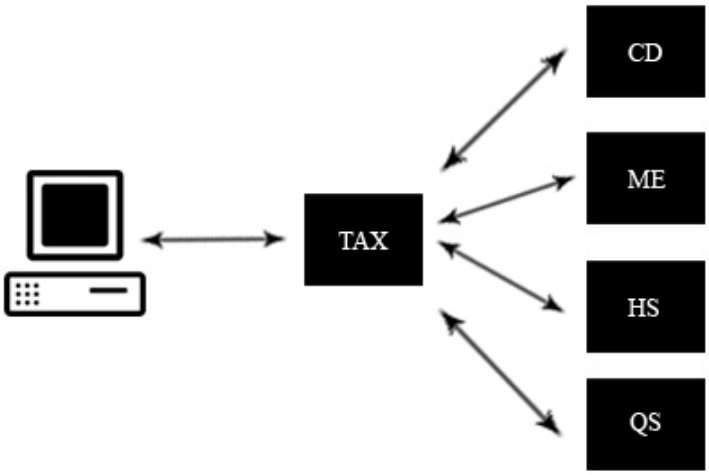

Figure 3.2: Main components of TRADExpress trading system (Adapted from ([8]))

3.3 Trading orders

3.3.1 Order Book

The order book is the core component of a trading system where the orders that are placed on an exchange by market participants are registered and displayed. The order book can be either public or private. In public order book the orders are visible to any market participant but anonymous to prevent misconduct, whereas the private order book is only visible to a specific participant and displays the orders placed by that very participant [7].

The order book's structure is divided into two sides; bid side and ask side that respectively present buy orders and sell orders (figure 3.3). The orders are sorted descending based on their priority in the order book. These orders are matched against each other. The matching mechanism is described in 3.4.

Bid side	Ask side
b_0: bid order with highest priority	a_0: ask order with highest priority
.	.
.	.
.	.
b_n: bid order with lowest priority	a_n: ask order with lowest priority

Figure 3.3: Order book structure

3.3.2 Orders Attributes

Various attributes could be attached to an order that combination of those make specific orders that meet the participant's preferences. Also these attributes show the order's priority in order book. The orders attributes come with their description in the following [7]:

- Side: The fundamental component that should be specified to recognize if the order is a sell or a buy order.
- Quantity: Shows the amount of the order that may be traded, it is possible to choose if the whole quantity of the order should be executed or partial filling is also accepted.
- Price Condition: Specifies which prices are accepted for trading and how they should be formed:
    - Market price: the order will be executed at the price set by the market.
    - Limit Price: the order will be executed if the trade price is as good as or better than a given one.
    - Pegged Order: the accepted execution price of the order depends on other prices.
- Visibility: Specifies if the order is transparent (visible in the public order book), or dark (visible only in the private order book).
- Validity period (Duration): Specifies the start and end of the period when an order is allowed to match

3.3.3 Order Types

Combining the order attributes of the section 3.3.2 provides different trading strategies for TRADExpress participants. Table 3.1 shows variety of the order types. P and q in the table stand for price and quantity respectively [7].

Type	Description
Market	Executed at the price set by the market and then cancelled.
Limit	Executed if the price is equal or better than p.
Fill-or-Kill (FoK)	Either executed at the whole quantity q when inserted or cancelled.
Fill-and-Kill (FaK)	Executed when inserted, the remaining quantity (if any) is cancelled.
All-or-none (AoN)	Executed only at the whole quantity q.
Iceberg	Executed considering the quantity q, but just showing the public quantity p_q.
Pegged	Executed at the best price in its side plus some price offset P_o. Cancelled if there are no orders to peg to.

Table 3.1: Trading strategies

3.3.4 Order Priority

The orders are prioritized in the order book regarding following attributes. The order with highest priority is eligible to be traded first [7]:

- Price: the order with the better price gets higher priority; better price for buy side is one with higher price and for the sell side is one with lowest price.
- Visibility: transparent (not dark) orders get higher priority
- Quantity: orders with minimum quantity get higher priority
- Time: orders inserted earlier in time get higher priority

The following example shows how orders are prioritized in buy side of the order book. From now on the following notions (Table 3.2) will be used to present different order types [20]:

Notation	Description
q @ p	A limit order with limit quantity q and limit price p
q ($\geq q_m$) @ p	A minimum quantity limit order with quantity q, minimum quantity q_m and limit price p
q ($\geq q$) @ p	An all-or-none order with quantity q and limit price p

Table 3.2: Different types of orders

The following example shows the prioritizing of orders in TRADExpress; suppose that the following bid orders are entered to the order book consecutively:

- 10 @ 9
- 10 (≥ 10) @ 10
- 10 @ 10
- 10 @ 10 (dark)

The orders will be prioritized like the table below:

Bid	Ask
b_0: 10 @ 10	
b_1: 10 (≥ 10) @ 10	
b_2: 10 @ 10 (dark)	
b_3: 10 @ 9	

b_0 gets higher priority as it has the better price in bid side; b_1 gets higher priority than b_2 because it is transparent order. b_2 gets higher priority than b_3 for the better price.

3.3.5 Trading States

The trading state can be either call auction or continuous. In call auction state the orders are accumulated during a period and at a specified time they are executed at a single price, the equilibrium price, which is set to the level that will result in the largest tradable volume. In

continuous state, the orders could be executed at any time. The incoming order is matched against the corresponding order; otherwise it is saved in order book till matching with another order [7].

3.4 Order Matching

As mentioned in previous section, there are two trading states in the system. In call auction state no trades are executed till the system goes to the continuous state, when a single price is calculated as an equilibrium price. At the equilibrium price, all the bid orders above the price and all ask orders below the price are traded; also maximum turnover is attained.

In continuous state, auto-matching is used as the matching mechanism. In auto-matching, an incoming order can be matched only against order(s) in another side of the order book. A limit bid order is matched against order(s) on the ask side which have equal or lower price than the limit bid price. The price of the trade is set by the resting order; it means that if two orders have the eligibility for matching, the price of the trade would be the price of the order that was resting in order book before another order entered.

3.5 EMAPI

EMAPI (external message application programming interface) is a proprietary interface that provides access to the TRADExpress trading system for clients. Through the EMAPI, with granted access, an application can access to all the information and all services [8]. EMAPI provides a huge number of messages for market management, data handling and etc. The focus of this project is applying delta debugging on messages that are communicated between the client and the TRADExpress trading system. To ease the report for readers, it is avoided to describe the messages in detail; in the next chapter some messages that have been used in the implementation and test of this project will be introduced in brief.

4 Delta Debugging

One of the steps in debugging a program is automating and simplifying the failing test case. After reproducing the failure, it is the time to simplify the test case and remove the circumstances that are not relevant to the failure. A circumstance is said to be relevant if it is required to make the failure and it is irrelevant if the failure occurs whether the circumstance is present or not. The Delta Debugging algorithm automatically simplifies a failing test case by successive testing till reaching a minimal test case that removing any single input from the test case causes the failure to disappear.

4.1 Definitions and Concepts

A circumstance is anything that can influence the execution of a program. A circumstance can include program code, its input, its code or its environment that causes the program execution. The changeable circumstances are those whose changes cause a different program behavior and in the remainder of this chapter, "circumstances" refer to changeable circumstances.

4.1.1 The change that causes a failure

A set of possible configurations of circumstances denoted by R. Each $r \in R$ determines a specific run of a program. To find the cause of a failure we focus on the difference between $r_{\times}$ and some $r_{\surd} \in R$ that works:

Definition 1 (Change) *A change δ is a mapping $\delta : R \text{ to } R$. The set of changes C is the set of all mappings from $R \rightarrow R$. The relevant change between two runs $r_1, r_2 \in C$ is a change $\delta \in C$ such that* $\delta(r_1) = r_2$

In the remainder of the chapter δ stands for the relevant change between two given program runs $r_{\times}$ and $r_{\surd}$.

4.1.2 Decomposing Changes

A relevant change can be decomposed to a number of elementary changes $\delta_1, \ldots\ldots, \delta_n$. This decomposition of δ into individual changes δ_i is program specific; it can be adding a tag in a HTML code or changing a line of program code. It is desirable to decompose a change as much as possible to reach the *atomic* decomposition that cannot be decomposed further.

Definition 2 (Composition of changes) *The change composition $\circ : C \times C \rightarrow C$ is defined as* $(\delta_i \circ \delta_j)(r) = \delta_i(\delta_j(r))$.

In practice, $\circ$ is realized as a *union* of two change sets δ_i.

4.1.3 Test Cases and Tests

Test function takes a program run and tests if it produces failure. This function may return one of the following results:

- The test succeeds (PASS, written here as √)
- The test has produced the failure (FAIL, written here as ×)
- The test produced indeterminate results (UNRESOLVED, written here as ?)

Definition 3 (*rtest*) *The function rtest:* $R \rightarrow \{\surd, \times, ?\}$ *determines for a program run* $r \in R$ *whether some specific failure occurs* $(\times)$ *or not* $(\surd)$ *or whether the test is unresolved (?).*

Axiom 4 (Passing and failing run) $rtest(r_i) = \surd$ *and* $rtest(r_\times) = \times$ *hold.*

The $c_\times$ is defined as the empty set $c_\surd = \emptyset$ which identifies $r_\surd$ (no changes applied). The set of all changes $c_\times = \{\delta_1, \delta_2, \ldots\ldots, \delta_n\}$ identifies $r_\times = (\delta_1 \circ \delta_2 \circ \ldots \circ \delta_n)(r_\surd)$. The subsets of $c_\times$ *test cases:*

Definition 5 (Test case) *A subset* $c \subseteq c_\times$ *is called a test case.*

Test cases are related to program runs by means of the test function, which applies the set of changes to $r_\surd$ and tests the resulting run.

Definition 6 (test) The function test: $2^{c_\times} \rightarrow \{\surd, \times, ?\}$ is defined as follows: Let $c \subseteq c_\times$ *be a test case with* $c = \{\delta_1, \delta_2, \ldots\ldots, \delta_n\}$. *Then* $test(c) = rtest((\delta_1 \circ \delta_2 \circ \ldots \circ \delta_n)(r_\surd))$.

Using Axiom 4, we can deduce the results of $test(c_\times)$ and $st(c_\surd)$:

Corollary 7 (Passing and failing test case) *the following holds:*
$test(c_\surd) = test(\emptyset) = \surd$ ("passing test case") and
$test(c_\times) = test(\{\delta_1 \circ \delta_2 \circ \ldots \circ \delta_n\}) = \times$ ("failing test case").

4.1.4 Minimizing test cases

The purpose of the simplification of a test case $c_\times$ is minimizing the difference between $c_\surd$ and $c_\times$. A test case $c \subseteq c_\times$ being a minimum means that there is no smaller subset of $c_\times$ that causes the test to fail.

Definition 8 (Minimal test case) *A test case* $c \subseteq c_\times$ *is minimal if* $\forall\, c' \subseteq c_\times$. $(test(c') \neq \times)$ *holds.*

In other words we want to minimize a test case such that removing any change causes the failure to disappear which means all circumstances are relevant in producing the failure. This goal is practically impossible to achieve as minimizing a test case requires testing $2^{|C|} - 1$ subsets of *c* which obviously has exponential complexity.

To make the theory come to use we rely on determining an approximation; removing a set of changes from the test case is still significant in producing the failure. This property is called *n- minimality*: removing any combination of up to *n* changes causes the failure to disappear.

Definition 9 (*n*-minimal test case) *A test case* $c \subseteq c_{\times}$ *is n-minimal if* $\forall c' \subseteq c_{\times} . (|c'| < |c| \Longrightarrow test(c') \neq \times)$ *holds.*

This means if a failing test case is *3-minimal*, removing any combination of 3 changes or less cause the test case to pass. Consequently we can define a *1-minimal* failing test case as a test case that excluding any single change from it cause the test case to pass.

Definition 10 (*1*-minimal test case) *A test case* $c \subseteq c_{\times}$ *is 1-minimal if* $\forall \delta_i \subseteq c . test(c - \{\delta_i\}) \neq \times$ *holds.*

The aim of simplification is achieving *1-minimality*, which requires an effective algorithm to reduce the number of tests are run to minimize the test case.

4.2 Simplifying the Problem

Recall to the beginning of this chapter, simplification is a method of turning a detailed problem report into a small failing test case in which every circumstance is relevant in producing the failure. Simplification helps to determine what details of the problem is relevant and what are not even if we do not have any clue about the cause of the failure. For instance taking crash of a plane after some minutes of its take off as the problem for simplification, we might take out the passenger seats or coffee machine and find that plane still crashes. We might take out engines and see that plane does not move, so engines are relevant to the crash.

Beside the finding failure causes, simplification brings three more advantages:

1. ***A simplified test case is easier to communicate.*** "The simpler a test case, the less time it takes to write it down, to read its description, and to reproduce it. In addition, you know that the remaining details are all relevant because the irrelevant details have been taken away".

2. ***A simplified test case facilitates debugging.*** "Typically, a simplified test case means less input (and thus smaller program states to examine) and less interaction with the environment (and thus shorter program runs to understand)".

3. ***Simplified test cases identify duplicate problem reports.*** "Duplicate problem reports can fill up your problem database. Simplified test cases typically subsume several duplicate problem reports that differ only in irrelevant details".

4.2.1 Manual Simplification

The method is sketched by Kernighan and Pike (1999) and results in a divide-and-conquer process:

> *Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.*

Manual simplification requires running a lot of tests; it takes effective time of the programmer for doing an unchallenging task. So it is desirable to automate the test case simplification.

4.2.2 Automatic Simplification

To automate the simplification we first need to implement a test function that decides if a subset of the test case fails or not. That is the way we distinguish if that part of the test case is relevant to the failure or it is not. Second, we need to implement a strategy which realizes the binary search by running the test on subset of circumstances.

To implement the second part we must:

1. Cut away half of the input and run the test. If test fails, continue the process with the remaining half.
2. Otherwise, go back to the previous state and discard the other half of the input.

What if none of the halves fail? Here the strategy mentioned earlier comes to use. Instead of testing the half of the input, run the test on smaller subset of the input such as its quarter, eighths, and sixteenths and so on.

4.3 Delta Debugging

The delta debugging algorithm is a general approach to simplify the failing test case; $ddmin'$ is a recursive function in core of the algorithm which gets two arguments: the configuration (input) to be simplified (denoted as c'_x) and the granularity n.

Depending on test result $ddmin'$may execute one of these three actions:

- Invokes itself with a smaller c'_x("some failing input")
- Invokes itself with double granularity ("increase granularity")
- Ends the recursion

$ddmin$ is a variant approach of the delta debugging which isolates failure causes by narrowing down differences (deltas) between runs and guarantees that the returned c'_x is 1-minimal. Figure 4.1 demonstrates $ddmin$ algorithm in a nutshell.

The test function *test(c)* of algorithm decides whether the test outcome is failed, passed or is unresolved; base on this result the algorithm decides if to continue with one of the failing halves of the input or it should increase the granularity n and split the input to smaller (quarters, eighths and so on) subsets [4].

- Let a program's execution be determined by a set of circumstances called a *configuration*. The set of all circumstances is denoted by $\mathcal{C}$.
- Let *test*: $2^{\mathcal{C}} \rightarrow \{\text{✗}, \text{✓}, \text{?}\}$ be a testing function that determines for a configuration $c \subseteq \mathcal{C}$ whether some given failure occurs (✗) or not (✓) or whether the test is unresolved (?).
- Let $c_{\text{✗}}$ be a "failing" configuration with $c_{\text{✗}} \subseteq \mathcal{C}$ such that $test(c_{\text{✗}}) = \text{✗}$, and let the test pass if no circumstances are present [i.e., test $(\emptyset) = \text{✓}$].
- The *minimizing delta debugging algorithm* $ddmin(c_{\text{✗}})$ minimizes the failure-inducing configuration $c_{\text{✗}}$. It returns a configuration $c'_{\text{✗}} = ddmin(c_{\text{✗}})$ such that $c'_{\text{✗}} \subseteq c_{\text{✗}}$ and $test(c'_{\text{✗}}) = \text{✗}$ hold and $c'_{\text{✗}}$ is a *relevant configuration*—that is, no single circumstance of $c'_{\text{✗}}$ can be removed from $c'_{\text{✗}}$ to make the failure disappear.
- The *ddmin* algorithm is defined as $ddmin(c_{\text{✗}}) = ddmin'(c'_{\text{✗}}, 2)$ with

$$ddmin'(c'_{\text{✗}}, n) = \begin{cases} c'_{\text{✗}} & \text{if } |c'_{\text{✗}}| = 1 \\ ddmin'(c'_{\text{✗}} \setminus c_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1 \ldots n\} \times test(c'_{\text{✗}} \setminus c_i) = \text{✗} \\ & \quad \text{(“some removal fails”)} \\ ddmin'(c'_{\text{✗}}, \min(2n, |c'_{\text{✗}}|)) & \text{else if } n < |c'_{\text{✗}}| \text{ (“increase granularity”)} \\ c'_{\text{✗}} & \text{otherwise} \end{cases}$$

where $c'_{\text{✗}} = c_1 \cup c_2 \cup \cdots \cup c_n$ such that $\forall c_i, c_j \times c_i \cap c_j = \emptyset \wedge |c_i| \approx |c_j|$ holds.

The recursion invariant (and thus precondition) for *ddmin'* is $test(c'_{\text{✗}}) = \text{✗} \wedge n \leqslant |c'_{\text{✗}}|$.

Figure 4.1: The minimizing delta debugging algorithm in a nutshell

4.4 Isolating Failure Causes

When we have a passing and a failing test case in hand, it is more efficient to use another approach to find the failure circumstances which is called *isolating*. The output of this approach is a pair of test cases; one passing test case and one failing test case with the minimal difference between them which is actual cause of the failure.

The process of narrowing down the failure circumstances is a boring, tedious and labor intensive work; with automating one can make this process fast and avoid mistakes that can occur during manual debugging. The requisites are:

- an *automated test* that checks whether the failure is still present,
- a means of *narrowing down the difference*, and
- a *strategy* for proceeding.

4.4.1 Simplifying versus Isolating

The output of the simplifying is a minimal test case in which every circumstance is relevant to the failure and removing any single circumstance makes the failure to disappear, whereas the isolating results in a passing and failing test case with the minimal difference which is the actual cause of the failure.

In isolating like what is done in simplifying, the smaller test case is used as the failing test case whenever the test case fails. In addition, the circumstances of failing test case are added to the passing test case, which may result to a larger test case. Figure 4.2 shows the difference between two approaches.

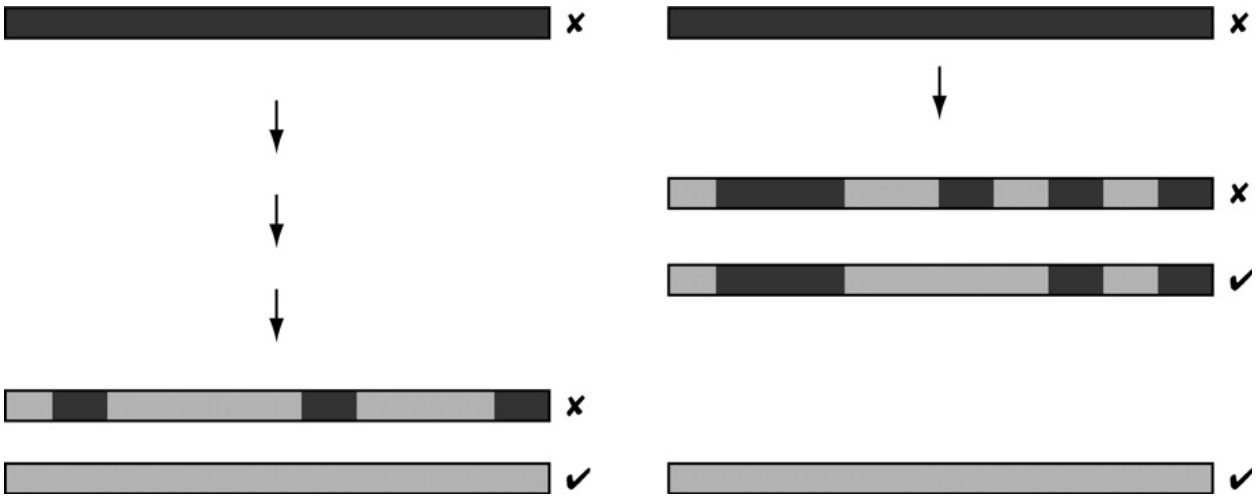

Figure 4.2: Simplifying versus isolating

Briefly we can distinguish the difference of simplifying and isolating with two following descriptions:

- ***Simplification*** brings the failing test case to a minimal test case where each circumstance is relevant in making the failure and removing any single circumstance makes the failure disappear.
- ***Isolation*** finds a relevant part of the test case that is relevant in making the failure.

4.4.2 An Isolation Algorithm

The original *ddmin* algorithm that described in section 4.4 can be extended to compute a minimal difference instead of a minimal test case. This extension is called *general delta debugging algorithm (dd)*. *dd* has the similar worst-case complexity as *ddmin*. If almost all tests have unresolved outcome, the number of tests can be quadratic $O(n)$, where n is cardinal number of the initial configuration. If more tests fail or pass, the algorithm becomes more efficient in number of tests. The complexity becomes up to logarithmic in the case that all tests have

resolved outcome; hence the goal of the *dd* algorithm is to keep the number of unresolved tests to a minimum [3]. Figure 4.3 shows the *general delta debugging* algorithm in a nutshell. You may refer to chapter 13 of [3] for more details about the algorithm.

$$dd_2(c, c', n) = \begin{cases} dd_2(c, c \cup c_i, 2) & \text{if } \exists c_i \cdot test(c \cup c_i) = \text{✘} \\ dd_2(c' - c_i, c', 2) & \text{else if } \exists c_i \cdot test(c' - c_i) = \text{✓} \\ dd_2(c \cup c_i, c', \max(n-1, 2)) & \text{else if } \exists c_i \cdot test(c \cup c_i) = \text{✓} \\ dd_2(c, c' - c_i, \max(n-1, 2)) & \text{else if } \exists c_i \cdot test(c' - c_i) = \text{✘} \\ dd_2(c, c', \min(2n, |c'| - |c|)) & \text{else if } n < (|c'| - |c|) \\ (c, c') & \text{otherwise} \end{cases}$$

where $c' - c = c_1 \cup c_2 \cup \cdots \cup c_n$ with c_i pairwise disjoint and $\forall c_i \, (|c_i| \approx (|c'| - |c|)/n)$.

The recursion invariant for dd_2 is $test(c) = \text{✓} \wedge test(c') = \text{✘} \wedge n \leq |c'| - |c|$.

Figure 4.3: The general delta debugging algorithm in a nutshell

5 Development

In this chapter the process of the automated debugger development from requirement analysis to implementation comes into detail; the requirement of the automated debugger will come first, then architecture of testing framework at Cinnober will be described and there will be a study of how the automated debugger should be integrated to the testing framework.

Design of the input, the debugger structure and the output are explained as well. The process of applying delta debugging in the model from failure detection to retrieving the actions which cause the failure comes after with an example that gives details of implementation as well.

5.1 Requirements

The developed automated debugger must fulfill the following functional and non-functional requirements:

Functional requirements:

- It must generate an execution profile[1] from simulation: execution profile is input of the model; automated debugger reads and runs the actions of execution profile.
- It must be able to reproduce the failure: the automated debugger has to reproduce the failure before beginning the debugging process.
- It must reproduce the failure deterministically: the attributes of transactions and their sequence when reproducing the failure should be the same as the failing test case.
- It must retrieve the relevant actions to failure: the causing failure action(s) is the output of the model.
- It must minimize the failing test case till reaching 1-minimality (Section 4.2.4, Definition 10)

Non-functional requirements:

- Automated debugger must not delimit test development: the development of automated debugger should be in a way that does not affect the testing development. Automated debugger should re-run any automate random test dependent on how it is written.
- Automated debugger should be as quick as possible

[1] Execution profile is the logged actions during the simulation with their attributes; for more information about execution profile look at section 5.5.1

5.2 Testing Framework

5.2.1 Testing framework architecture

The layered architecture of Cinnober's testing framework is illustrated in figure 5.1. As shown in the figure, Cinnober's testing framework extends the JUnit, standard Java testing framework and itself is divided into two sub-frameworks: FIX and EMAPI; these frameworks facilitate developers and testers to start FIX and EMAPI sessions with TRADExpress system and perform deterministic tests and verifications. The both functional and non-functional tests can be automated in the framework.

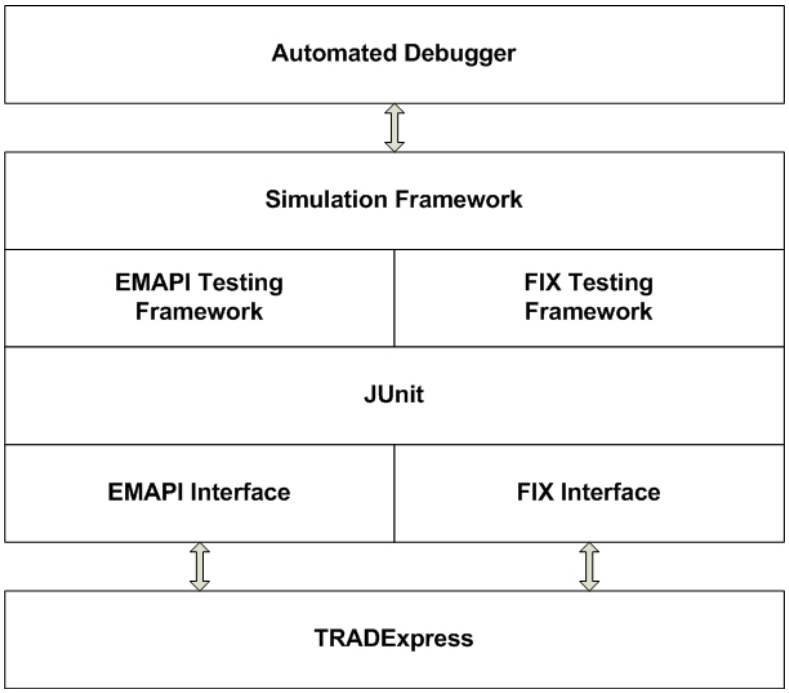

Figure 5.1: Testing framework architecture

5.2.2 Random testing framework

The random testing framework is layered on top of the testing framework; it provides the tools for generating random test cases in FIX and EMAPI protocols. The random test framework is basically composed of three components:

- Actor: An actor is the collection of the actions; selection of an action depends on the probability that is considered for that action. An action could be any order entry, update or cancel.
- Oracle: An oracle is a mechanism to verify if the output of the system under test is the one expected based on the input.

- Simulation: Simulation is the core component which controls the simulation process. It selects the actor and executes the corresponding action. In addition it controls the duration of the simulation which could be based on time duration or simulation steps.

To simulate an action, the core simulation performs the following steps:

- The actor that should perform its action is selected by simulation core
- The actor chooses one action that it can take
- Oracle that is triggered by the chosen actor, calculates the expected result
- If the expected result is equal to the actual result the simulation continues, otherwise the failure is reported to the simulation core and simulation will be stopped.

5.3 Automated debugger integration

As the core simulation is responsible to execute the actions, the automated debugger requires access to the simulation component for reproducing the actions of the failing test case. This implies that the automated debugger should be layered on top of the testing framework and to extend the simulation component (Figure 5.1).

5.4 Design specifications

In this section the code structure of the automated debugger is presented with the explanation of each class, template of the input and how it should be passed to the debugger and what we expect as an output comes in detail. In addition, the procedure of debugging a failed test case from failure detection to debug is given.

5.4.1 The input

The input to the automated debugger is a failing test case. How the input should be fed to the automated debugger? One option is to run the failed test case and reproduce the failure. Suppose that there are hundred actions in a test case, and the test case fails after 80^{th} action. If we run the whole test case it results in more action runs and consequently more debugging time. Therefore, the better solution is to run test case up to the action that makes the failure. So we need to keep track of the actions that are already run; to do so each action should be logged before execution and logging of actions should be stopped as soon as failure occurs.

The input to the automated debugger is called execution profile; the template of the execution profile and how it is created is presented in section 5.5.1.

5.4.2 The output

The output of the automated debugger is the 1-minimal failing test case which contains exact actions that cause the failure; the output template is same as the input template. For debug purpose, developers can run the output without going through sequence of a lot of actions since they have absolute actions that make the failure.

5.4.3 Structure of the automated debugger

In this section, first the code structure of the automated debugger is introduced and then responsibility of each class is explained. In section 5.6 there is an example with the sequence diagram of the automated debugger that makes the understanding of the automated debugger procedure more clear. Figure 5.2 illustrates UML class diagram of the automated debugger. The diagram shows the relation and multiplicity between classes. *TestCaseReproducer* and *DeltaDebugger* are core classes of the automated debugger. The role of each class is explained below.

Simulation class

This class is the main class in random test framework which starts the simulation and has following tasks:

- Control simulation duration
- Select the actor who should act
- Get the action that should be executed from the actor
- Print simulation information

TestCaseReproducer class

TestCaseReproducer inherits Simulation class to be able run the simulation, but it does not perform the same tasks. This class is responsible to recreate actions read from the execution profile. This class basically uses *java.lang.reflect* package to construct classes and call methods that are logged in execution profile. This class is generic and is independent from implementation of actors. Consequently, there is no restriction for test developers that they should observe to have compatible tests with the model.

DeltaDebugger class

DeltaDebugger class is the core class of automated debugger which controls the procedure of debugging. Hence, it aggregates *ActionLogReader*, *LogParser* and *TestCaseReproducer* for purpose of reading execution profile, checking for failure and generating actions respectively.

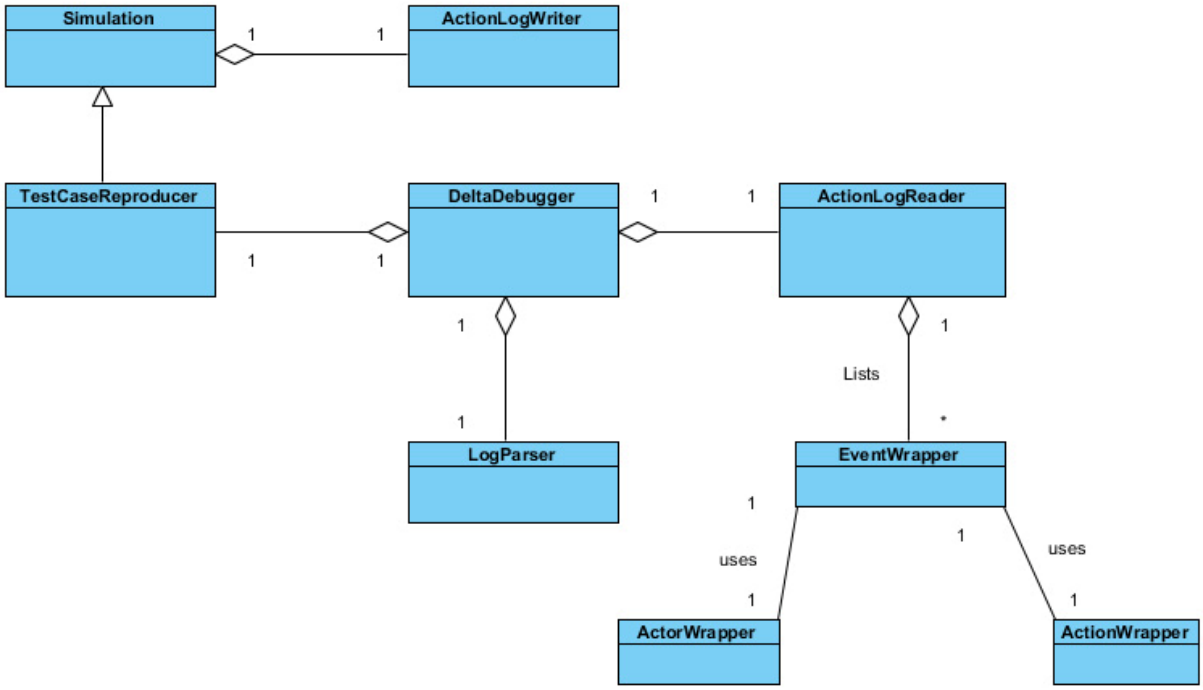

Figure 5.2: UML diagram of delta debugging

Delta debugging algorithm is implemented in this class and the decision of which subset of actions should be executed is made by this class. *DeltaDebugger* instances *TestCaseReproducer* to execute the actions. After execution of each subset, *LogParser* checks the log files of the system under test for any failure and based on the failure of any subset, the next subset is selected.

ActionLogWriter class

This class is responsible to create the execution profile and implements several methods for logging actors and actions and also method attributes. As shown in class diagram, Simulation class creates an instance of this class to log the actions. *Simulation* class calls *log* method of *ActionLogWriter* after actor and action selection to write the action. In section 5.5.1 the template of the execution profile is described.

EventWrapper

As its name shows, this class encapsulates actor and action. An event comprises of an action and an actor that depicted in Figure 5.2.

ActionLogReader class

This class reads the execution profile and creates the list of events. Event list is used by *TestCaseGenerator* to reproduce the test. *DeltaDebugger* is a mediator between two classes; it gets the list of events at the begging of the debugging from *ActionLogReader* and passes the subset of events to *TestCaseGenerator* to be executed.

LogParser class

This class receives the name of the servers that should be checked for any failure and parses the corresponding log files. It is also possible to configure the parser to search for a list of failures.

5.5 Procedure of delta debugging

The flowchart in Figure 5.4 shows the procedure of delta debugging in TRADExpress. The procedure starts with running and then logging actions. Immediate after running of tests, servers' log files like ME, CD, TAX and etc are parsed for any failure. If any failure encountered, the actions are reproduced and delta debugging is performed on actions to retrieve those are relevant to the failure.

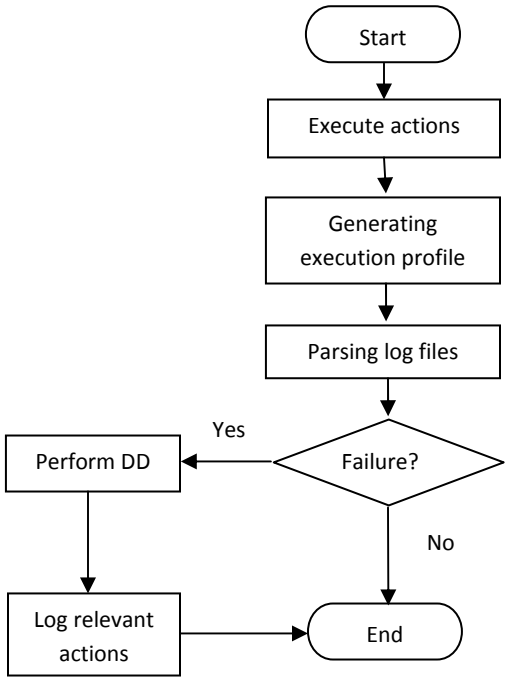

Figure 5.3: Procedure of delta debugging in TRADExpress

5.5.1 Generating execution profile

In simulation framework following steps are pursued to generate a simulation cycle:

- Actor that should perform its action is selected by simulation core
- Actor chooses one action that it can take
- Action is executed by the simulation core

To be able to reproduce the same sequence of actions with the same attribute values, we should save simulation random seed. To do so, a file is created at the start of simulation which is called *execution profile*. After execution of each message, actor and action are logged in this file. Figure 5.3 shows the *execution profile* template.

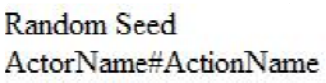

Figure 5.4: Execution profile template

Random Seed is simulation's random number; simulator utilizes this number to generate actions with the same order and type of actions. *ActorName* is name of the actor selected by simulator to perform its action and *ActionName* is name of the action or message that is performed.

5.5.2 Parsing log files

The time of parsing log files has a high effect on duration of the delta debugging procedure which explained in previous section. Since the log files getting bigger and bigger in size after running each test case, parsing them at the end of the random testing takes a long time and increases procedure duration. To make the debugging procedure faster, log files are parsed simultaneously with running tests. In this approach, number of the lines has been read should be kept to be skipped next time the log file is parsed. So due to existence of several log files there should be a mapping between every log file and number of lines read from that log file.

5.5.3 Performing delta debugging

If any test case fails, the control of the debugger is transferred to the core class of the automated debugger *DeltaDebugger* class. DeltaDebugger gets the logged actions from *ActionLogReader* and passes it to *TestCaseReproducer* to re-generate the actions. It is DeltaDebugger class that decides which subset of actions should be reproduced. After generating actions, *LogParser* parses system's log files to check if any exception is thrown. DeltaDebugger always continues the debugging with the failed subset.

5.5.4 Log relevant actions

At the end of the debugging, actions of the last subset which has 1-minimality property are written in a file. Developers use these actions to reproduce and fix the bug.

5.6 Case study

In this section a TRADExpress case study will be analyzed from failure detection to test case minimization. The process of the delta debugging on test case which were explained in section 5.5 will be applied step by step on this test case.

5.6.1 Test case

The figure 5.5 shows a random test case that generates thousand actions. At the beginning of the test case random seed and length of simulation time are set respectively.

Actions are weighted and base on the assigned weights actor chooses one of the actions to execute. At the end of the test case simulation starts by calling the startSimulation method.

```
public void reproduceFailure() throws Exception {
    setSeed(7483738L);
    setExecutionLength(1000L);
    OrderBookViewer tViewer = new OrderBookViewer(this, USER, mInstrument);
    addOracle(tViewer);
    addActor(new CompositeActor(this, "ProduceFailure", USER, mInstrument)
        .useOrderBookViewer(tViewer)
        .weightAction(CompositeActor.CREATE_ORDER, 40)
        .weightAction(CompositeActor.UPDATE_ORDER, 40)
        .weightAction(CompositeActor.SWITCH_HALT, 10)
        .weightAction(CompositeActor.CANCEL_ORDER, 10));

    startSimulation();
}
```

Figure 5.5: The random test case

5.6.2 Logging actions

Before action execution, it is logged in execution profile. The template of execution profile is depicted in section 5.5.1. A part of generated execution profile from the above random test case is presented in figure 5.6.

The number at the beginning of execution profile is the random seed.

```
7483738
com.cinnober.emapi.simulator.impl.CompositeActor#@createOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@createOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@updateOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@cancelOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@createOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@createOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@updateOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@switchHalt%
com.cinnober.emapi.simulator.impl.CompositeActor#@createOrder%
com.cinnober.emapi.simulator.impl.CompositeActor#@updateOrder%
```

Figure 5.6: A part of execution profile

5.6.3 Performing delta debugging

When a test case fails, it is time to start delta debugging to find relevant actions to failure. Figure 5.7 presents sequence diagram of automated debugger. First of all execution profile is read by ActionLogReader class and actor and corresponding actions are saved in ActorList and ActionList. Both lists are encapsulated in EventList. EventList is read by DeltaDebugger class to calculate which subset of actions should be run.

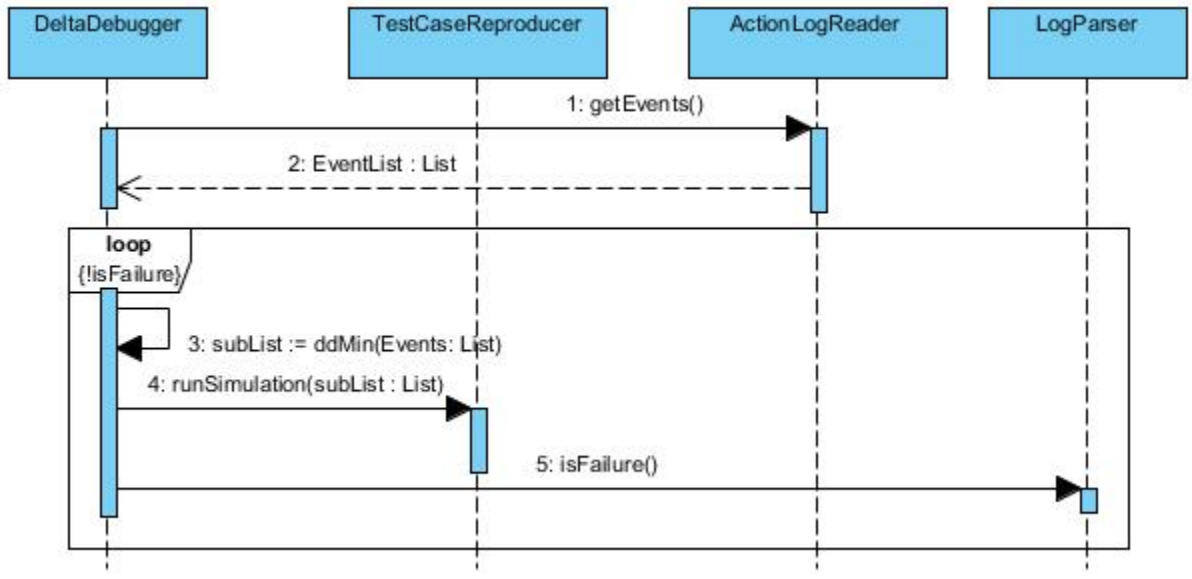

Figure 5.7: Sequence diagram of delta debugging

The decision is made by ddMin method; ddMin method is the main method in the DeltaDebugger that implements the delta debugging algorithm. Then the subList returned by ddMin is passed to runSimulation method of TestCaseReproducer class.

TestCaseReproducer class runs the actions of subList. DeltaDebugger checks if the subset fails or not by calling the isFailure method from LogParser class. Always the failed subList is chosen by DeltaDebugger for next calculation. This loop continues till the action list cannot be minimized further. The very last list is 1-minimal and is written in the output file as the result of the delta debugging.

6 Experiment and Evaluation

In this chapter two types of experiments will be presented. First, in section 6.1 the model is tested with real failed test cases that have been reported in the test management system of Cinnober, to verify the correctness of the model. In section 6.3, artificial failed test cases are generated to evaluate the performance of the system. By artificial failed test cases, we will evaluate how changes to the different parameters of a failed test case affect the performance of the automated debugger model.

In Section 6.4, it is discussed how the performance of the model can be enhanced by applying some test reduction methods. The experiments of section 6.3 are repeated in this section to show how well the methods have affected the performance.

At the end, there is a discussion about what is achieved by developing the automated delta debugger and how it satisfies the testing policy of Cinnober AB.

6.1 Real failed test cases

To verify the correctness of the model's functionality, we should have the output (the root cause of the failure) in hand to compare it with the automated debugger's output. The test management system was a good source to use the already reported and fixed bugs. The model was tested with several reported bugs and the output comparison showed that the model functions correctly and returns the minimal test case as the output. Since we do not want to go in detail of every bug, one of the failed test cases is presented below.

Figure 6.1 shows the failed test case; this test case is given as the input to the automated debugger model to retrieve the relevant actions that cause the failure. The test case generates randomly hundred actions that includes the following five action types:

- INSERT_ALL_OR_NOTHING: inserts an order with quantity q and price p. The order executes only at quantity q.
- INSERT_ORDER: inserts an order with quantity q and price p.
- UPDATE_ORDER: randomly selects an order and updates its quantity and price.
- SWICTH_HALT: it halts or lifts the order book every time it is called. When an order book is in halt status, it is not possible to enter, update and cancel order in the order book. When the order book is lifted, it returns to the continuous state (section 3.4).
- MASS_ORDER_CANCEL: cancels all of the orders in the order book.

```
@Test
public void mainScenario() {
   setSeed(1268142L);
   setExecutionLength(100L);
   OrderBookViewer tViewer = new OrderBookViewer(this, USER, mInstrument);
   addOracle(tViewer);

   addActor(new CompositeActor(this, "TestComposite", USER, 100, mInstrument)
      .useOrderBookViewer(tViewer)
      .weightAction(CompositeActor.INSERT_ORDER, 30)
      .weightAction(CompositeActor.INSERT_ALL_OR_NOTHING, 20))
      .weightAction(CompositeActor.SWITCH_HALT, 30))
      .weightAction(CompositeActor.UPDATE_ORDER, 10)
      .weightAction(CompositeActor.MASS_ORDER_CANCEL, 10);

   startSimulation();

}
```

Figure 6.1: The random failed test case

At the end of the debugging, the model returns the following four actions as the output:

- INSERT_ALL_OR_NOTHING
- SWICTH_HALT
- SWICTH_HALT
- MASS_ORDER_CANCEL

Executing these actions sequentially reproduces the same failure and this is the minimal test case that we were expecting.

6.2 Artificial failed test cases

We need more tests to verify that the model functions properly under heavy load. Since the real failed test cases are not sufficient to evaluate the model concerning the performance, the artificial failed test cases are generated.

To generate the artificial failed test cases, some actions are defined as fake actions that together produce a failure when executed sequentially. The source code (matching engine server's source code) is manipulated to throw an exception when the number of executed fake actions reaches to a specified number. Figure 6.2 shows the pseudo code of TestCaseReproducer class that asks the matching engine to generate the failure.

```
1 int NUMBER_OF_ACTIONS = 2;
2
3  if (actionToBeExecuted == "FakeAction"){
4    nrOfExecutedFakeActions++;
5    if (nrOfExecutedFakeActions == NUMBER_INDUCING_ACTIONS) {
6              sendMessageToME("ThrowException");
7    }
8 }
```

Figure 6.2: TestCaseProducer asks matching engine to generate an exception

If the executed action is FakeAction (Line 3) then the nrOfExecutedFakeActions is increased by one (Line 4). Then it is checked if the nrOfExecutedFakeActions is reached to the specified number of actions (Line 5) if so, then a message is sent to the matching engine to aware it about the failure generation (Line 6).

Figure 6.3 shows the pseudo code of matching engine for failure generation. In line 2 if the received message is equal to "ThrowException", it means that the message is received from TestCaseReproducer and an exception should be generated (Line 3).

```
1 public void handleMessage(Message pMessage) {
2       if (pMessage.text == "ThrowException") {
3         throw ArtificialException("ArtificialFailure");
4       }
5       else
6       {
7         //Hanlde other type of messages
8       }
9 }
```

Figure 6.3: Matching engine throws exception upon receiving the message

We can evaluate how changing the number of the relevant failure circumstances (NUMBER_OF_ACTIONS in figure 6.2) affects the number of the tests in delta debugging algorithm and consequently duration of the debugging in the model.

6.3 Experiments and Analysis

The goal of the experiments in this section is to evaluate the effect of the following three parameters on the performance of the delta debugger model:

1. Size of the failing test case
2. Number of the failure circumstances
3. The location of the failure circumstances

6.3.1 Size of the failing test case

Table 6.1 shows the effect of the test case size on the performance of the delta debugger model. The experiments are categorized to different groups. In each group the size of the test cases are changed from 10 to 40 actions with 10 intervals and the number of failure circumstances remained constant. It seems the size of the input is in a direct relation with the number of the tests in every group, since with the increase of the first variable the other is raised as well. This is obvious since more actions mean more subsets and consequently more number of tests.

Group	Number of Actions	Number of failure-inducing actions	Index of failure-inducing actions	Number of tests
1	10	3	3, 6, 9	34
	20	3	6, 12, 18	44
	30	3	10, 20, 30	56
	40	3	13, 26, 39	63
2	10	4	2, 4, 6, 8	43
	20	4	5, 10, 15, 20	66
	30	4	7, 14, 21, 28	78
	40	4	10, 20, 30, 40	88
3	10	5	2, 4, 6, 8, 10	47
	20	5	4, 8, 12, 16, 20	77
	30	5	6, 12, 18, 24, 30	95
	40	5	8, 16, 24, 32, 40	105

Table 6.1: Effect of the test case size on the performance

6.3.2 Number of the failure circumstances

In table 6.2 the test cases are grouped with variable number of the failure circumstances and fixed size of the test case. In every group the number of tests is increased by raising the number of the failure circumstances. The more the number of failure circumstances, the more tests should be run.

Group	Number of Actions	Number of failure circumstances	Index of failure circumstances	Number of tests
1	10	3	3, 6, 9	34
	10	4	2, 4, 6, 8	43
	10	5	2, 4, 6, 8, 10	47
2	20	3	6, 12, 18	44
	20	4	5, 10, 15, 20	66

	20	5	4, 8, 12, 16, 20	77
3	30	3	10, 20, 30	56
	30	4	7, 14, 21, 28	78
	30	5	6, 12, 18, 24, 30	95
4	40	3	13, 26, 39	63
	40	4	10, 20, 30, 40	88
	40	5	8, 16, 24, 32, 40	105
5	100	6	16, 32, 48, 64, 80, 96	173
6	200	6	33, 66, 99, 132, 165, 198	214
7	300	6	50, 100, 150, 200, 250, 300	251

Table 6.2: Effect of the number of the failure circumstances

6.3.3 The location of the failure circumstances

Table 6.3 pictures the effect of the failure location on number of the tests that should be taken by the delta debugger model. *Subsets* column shows how delta debugging splits the actions. Each number in the column presents an action: the number in bold in each test case shows the failing action which cause the whole test case to fail. As shown in the delta debugging algorithm (Figure 4.1), the division of the actions starts with two subsets and the algorithm continues with the failing subset (or a complement of a subset). The shaded sub-columns show the subsets that are executed after each grouping.

Test Case	Subsets		Number of the tests
1	**1** 2 3 4	5 6 7 8	3
	1 2	3 4	
	1	2	
2	1 **2** 3 4	5 6 7 8	4
	1 **2**	3 4	
	1	**2**	
3	1 2 **3** 4	5 6 7 8	4
	1 2	**3** 4	
	3	-	
4	1 2 3 **4**	5 6 7 8	5
	1 2	3 **4**	
	3	**4**	
5	1 2 3 4	**5** 6 7 8	4
	5 6	7 8	
	5	-	
6	1 2 3 4	5 **6** 7 8	5
	5 **6**	7 8	
	5	**6**	
7	1 2 3 4	5 6 **7** 8	5
	5 6	**7** 8	
	7	-	

8	1 2 3 4	5 6 7 **8**	6
	5 6	7 **8**	
	7	**8**	

Table 6.3: Effect of failure location on number of tests

The first four test cases present the failure in the first subset of the test case, and the other test cases (4 to 8) present the failure in the second subset of the test case. Comparing first four test cases shows an increase in the number of the tests by changing the location of the failure from the very beginning of the subset to the end of the subset. The same thing happens in the second failing subset (test numbers 4 to 8).

If you compare the number of the tests between test numbers 1 and 4, 2 and 5 and so on, you see one test difference between them. This difference is the result of testing the first subset of the test case when the failure occurs in the second subset of the test case.

To sum up, it is proved that the delta debugging performance benefits from the failure that occurs early in the test case. This failure includes both the location of the failure and earlier failure in the subsets.

6.4 Performance Issues

The performance of the delta debugger model is evaluated based on the number of the executed tests on the failed test case to isolate the relevant actions that are the root cause of the failure.

The size of the test cases in the experiments of section 6.3.1 and 6.3.2 are not too large. Sometimes, the size of a test case can reach to more than several thousand actions. Therefore, appropriate approaches should be employed to decrease the number of the tests. The reason that the performance is evaluated based on the number of the tests is that, different transactions take different times to be completed; and also execution environment and system load affects the duration of the execution.

The following approaches can enhance the performance of the delta debugging [5]:

1. Skip the already passed set of actions
2. Use the *monotony* property: If a set of transactions pass the test, all subsets of that set will pass the test as well.
3. Reduce to the first failing subset if there are multiple dependent failure causes

As said before, we are interested in failing sets and there is no benefit in running set of actions that are already passed. Furthermore, by continuing with the first failing set we increase the probability of removing upcoming interrelated failures in the test case.

These methods dramatically increase the performance of the delta debugging. The experiments that are illustrated in table 6.2 are after applying the optimizations. Table 6.4 shows the result of

the same experiments before applying the optimizations. Comparing table 6.2 and 6.4 shows huge difference between the number of tests before and after optimization.

When the size of the failed test case is too huge even above optimizations cannot help to reduce the number of the tests. In this case, one can specify a threshold on the debugging duration or the number of tests it should execute; this strategy at least shortens the size of the test case and eases the manual debugging.

Row	Number of Actions	Number of failure-inducing actions	Index of failure-inducing actions	Number of tests
1	10	3	3, 6, 9	55
	10	4	2, 4, 6, 8	72
	10	5	2, 4, 6, 8, 10	77
2	20	3	6, 12, 18	83
	20	4	5, 10, 15, 20	121
	20	5	4, 8, 12, 16, 20	144
	20	6	3, 6, 9, 12, 15, 18	172
3	30	3	10, 20, 30	95
	30	4	7, 14, 21, 28	144
	30	5	6, 12, 18, 24, 30	179
	30	6	5, 10, 15, 20, 25, 30	225
4	40	3	13, 26, 39	110
	40	4	10, 20, 30, 40	166
	40	5	8, 16, 24, 32, 40	212
	40	6	6, 12, 18, 24, 30, 36	253
5	100	6	16, 32, 48, 64, 80, 96	359
6	200	6	33, 66, 99, 132, 165, 198	447
7	300	6	50, 100, 150, 200, 250, 300	536

Table 6.4: Experiments result before optimization

7 Discussion and further work

7.1 Discussion

Having presented the results of my experiments, it is valuable to assess its degree of reliability and applicability through analyzing its status in relevant theories and practice.

Through relevant literature reviews, it is concluded that delta debugging is the only method for minimizing a failing test case and other proposed methods are built on top of this algorithm. In fact, delta debugging can be applied on different inputs such as code, threads, variables and other system components.

Concerning the current status of the model, as it was mentioned earlier, delta debugger model is designed and developed for random testing framework of TRADExpress system in Cinnober Company. Since this system is a platform for all the customer projects in Cinnober Company, delta debugger model can be utilized in all projects with appropriate customization to be adopted with the specific trading strategies.

Another possible applicability of this model in Cinnober Company could be in regular JUnit testing. In some cases I have been asked if the model can debug a failure in the project, which is not produced in random testing and is just a typical JUnit test in the project? I should argue that the problem of debugging such test cases with delta debugger model is generating execution profile, which is the main requirement for reproducing failure. Since execution profile is generated by simulation component - the core component of random testing framework- we should think about another approach for logging transactions that would not introduce new, significant changes in the model.

One suggestion could be a component that lies between the sender and receiver servers in order to log the message. This new component would cause a delay in action execution that affects the system performance, but not its functionality. Generalizing the model could be the future work towards launching the model into practice.

There is an important limitation with using delta debugging model that is worth to be mentioned. One property of delta debugging that can prolong the debugging process and spoils automation process is its limitation in finding multi-independent errors. Delta debugging algorithm can find one error per run. In fact, for further failure findings, first, identified errors should be removed and then model should continue with new test case. Assume that testing test case *C* fails because of two independent changes C'_1 and C'_2. After first run C'_1 is found by delta debugger model; then model should continue with $C = C / C'_1$.

Another limitation that I faced in this project was the fact that it is not possible to estimate the time saving possibility and profitability of this model compared to manual debugging.
In fact, it is impossible to measure exactly how much time for a typical test case is saved using automated debugger model, since evaluations in previous chapter proved that time of debugging depends on several factors. However, through a quick assessment performed by an expert, it is possible to estimate the complication of the failure, its urgency for bug fixing and approximate time needed.

In general, it is always beneficial for financial companies with complicated systems like Cinnober to have an automated system for dealing with repetitive tasks. We cannot underestimate the fact that when the failure is not too complicated, the expert developers with good knowledge about the code can find the root cause of the failure faster than the automated debugger model. However, when the scale of system is considerably large, project deadline is close, human resource is not sufficient and failure is estimated to be highly complicated, then delta debugging system is one of the most practical solutions for our case. Since it reduces a considerable amount of developers' effort while puts the whole time-consuming, tiresome, error-prone debugging task on computers' capability.

7.2 Further work

As mentioned earlier, debugging large size inputs are too time-consuming even by delta debugger model. The future work will focus on speeding up the model. This performance enhancement requires a statistical investigation on the structure of the input to find out which combination of transactions has higher probability to fail. So, subsets with higher probability are tested first to increase the probability of getting a failing subset.

References

[1] Osro, Alessandro, et al. 2006. *Isolating Relevant Component Interactions with JINSI*. Shanghai, China. International Workshop on Dynamic Analysis (WODA 2006).

[2] Leitner, Andreas, et al. 2007. *Efficient Unit Test Case Minimization.* New York. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.

[3] Zeller, Andreas. 2009. *Why Programs Fail.* s.l. : Morgan Kaufmann Publications, 2009.

[4] Zeller, Andreas. 2001. *From Automated Testing to Automated Debugging.* 2001, IEEE Computer.

[5] Zeller, Andreas and Hildebrandt, Ralf. 2002. *Simplifying and Isolating Failure-Inducing Input.* s.l. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2002.

[6] Zeller, Andreas. 2002. *Isolating Cause-Effect Chains from Computer Programs*. Charleston, South Carolina, November 2002. Proc.ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10).

[7] Cinnober Financial Technology AB. 2009. An Introduction to Financial Markets. 2009.

[8] Cinnober Financial Technology AB. 2009. EMAPI Java Programmer's guide. 2009.

[9] Cinnober Financial Technology AB. 2009. TRADExpress Trading Engine Architecture. 2009.

[10] Cinnober Financial Technology AB. 2009. TRADExpress Trading System - System Overview. 2009.

[11] Cinnober. 2010. *http://www.cinnober.com/tradexpress-platform*. [Online] 2010. http://www.cinnober.com

[12] Carlgren, David. 2007. *Random testing of a market place system.* 2007 : Master's Thesis, KTH Royal Institute of Technology, 2007.

[13] Graham, Dorothy, et al. 2006. *Foundations of Software Testing: ISTQB Certification.* s.l. : Thomson Business Press, 2006.

[14] Hoffman, Douglas. 2001. *Using oracles in test automation*. Software Quality Methods. [Online] 2001. http://www.softwarequalitymethods.com/Papers/Auto%20Paper.pdf.

[15] Hoffman, Douglas. 1998. *A Taxonomy for Test Oracles.* California : s.n., 1998. Software Quality Methods, LLC.

[16] Misherghi, Ghassan and Su, Zhendong. 2006.*HDD: hierarchical delta debugging.* New York. ACM, 2006. Proceedings of the 28th international conference on Software engineering .

[17] Gundemark, Johan. 2005. Random tests in a market place system. Uppsala : Master's thesis, Uppsala University, 2005.

[18] Choi, Jong-Deok. Zeller, Andreas. 2002 Isolating Failure-Inducing Thread Schedules; Rome, Italy, July 2002. Proc. International Symposium on Software Testing and Analysis (ISSTA 2002).

[19] Höjeberg, Noah. 2006. *Random tests in a trading system.* Stockholm : Master's thesis, KTH Royal Institute of Technology, 2006.

[20] Castañeda, Roberto. 2010. *Constraint programming for random testing of a trading system.* Stockholm : Master's thesis, KTH Royal Institute of Technology, 2010.

[21] SmartBear. 2011. Why automated testing. *SmartBear.* [Online] 2011. http://www.automatedqa.com/products/testcomplete/manager-overview.