1-1-2010

A Study On Automatic Software Quality And Reliability Anlaysis

Ahmad Hosseingholizadeh
Ryerson University

A STUDY ON AUTOMATIC SOFTWARE QUALITY AND RELIABILITY ANALYSIS

by

Ahmad Hosseingholizadeh

Bachelor of Science, Shahid Beheshti University, Iran, 2008

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2010

Declaration

A STUDY ON AUTOMATIC SOFTWARE QUALITY AND RELIABILITY ANALYSIS

Master of Science 2010

Ahmad Hosseingholizadeh

Computer Science

Ryerson University

Abstract

In this thesis the study over the topic of analytical approaches for software quality and reliability assurance is presented. The focus of this research is on a specific set of techniques used for software reliability assessment called Risk Analysis. Numerous approaches are explored and different new techniques are proposed to generate the risk model of a software product. These techniques are evaluated and using the results of this evaluation a new risk model (Compound Risk Model) is proposed which is using the advantages of different classes of risk analysis techniques to generate a more precise and practical model to identify more risky components of a software product. Also a research on the topic of Automatic Bug-Fix using Genetic Programming is presented which can fix logical defects of a buggy code and evolve it to a bug-free code. Finally it is discussed that these approaches can be used as an automated tool in an integrated development environment to localize the defective components and debug them.

Acknowledgements

I am deeply thankful to my supervisor, Dr. Abdolreza Abhari, whose guidance, encouragement and support enabled me to develop the ideas of this thesis. His assistance throughout this study was abundantly helpful and made this work possible.

I'm grateful to Dr. Alexander Ferworn for his teachings that enlightened the path of productive study for me in the Master's degree.

I wish to thank Dr. Marcus Santos for his teachings which attracted me to the very interesting field of Genetic Programming.

I'm grateful to Dr. Alireza Sadeghian for his continuous and timely support during the course of my study.

My warm thanks to Dr. Cherie Ding for her excellent and practical teachings in the area of Service Oriented Architecture.

I'm thankful to all advisory committee members for their comments and guidances to edit and improve this work.

Finally, I take this opportunity to express my profound gratitude to my beloved family and friends for their moral support and patience during my study in Ryerson.

Contents

List of Tables

List of Figures

Chapter 1

Motivations and Objectives

In this chapter the area of the study will be introduced. The motivation, problem statement and the goal of this study in the area of software reliability will be presented and a foundation for the detailed discussion of the topic of this thesis will be provided.

1.1 Rationale

Software development is a complicated process. It involves many different factors and elements which have to be controlled and supervised to assure the quality of the final product. Many different methodologies and approaches have been proposed to simplify this process and make it more manageable. Each one of these approaches try to focus on one or some aspects of software development process; aspects such as:

- Project time estimation
- Required human resources
- Software product structure

- Required technologies and tools
- Quality assurance
- Feasibility study

Proper use of these techniques increases the chances of having a well managed and controlled project which is developed according to the estimated required time and meets the required quality. Quality of the final product is one of the most important factors of a software product which has a considerable effect on the success of the project in the market. From one point of view the quality of the product is a subjective measure, meaning that different users have different opinions about product's quality. But from another point of view the elements of quality can be identified and classified; elements such as reliability, efficiency, portability, usability, maintainability, testability, etc. (as *Cox* discussed in [2]). Out of this list reliability is one of the most important factors. The importance of reliability could be seen better when we know that according to statistical study by *Tassey* [3] only in US every year $20 billions could be saved if a better test is done before the final release of a new software product .

1.2 Nature of the Problem

This research is based on software reliability and correctness. Out of different techniques to increase the software reliability the focus of this study is on risk management and defect identification and correction. Different proposed techniques will be explored and by identifying their advantage and disadvantages suggestions are made to increase their precision and applicability. Also the results of using the proposed techniques in test environments will be presented and the results of these experiments will be discussed.

In the following section some key concepts of the area of this research will be defined and their applications will be explained.

1.2.1 Software Risk

According to *NASA Safety Manual* [4] risk is defined as the function of the possible frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with this frequency and severity. According to *Tao* [5] Risk in the context of software engineering is defined as the probability that a software development project experiences unexpected and inadmissible events such as termination, wrong budget and time estimations, poor quality of the software solution, wrong used technology, etc. The purpose of software risk management as described by Boehm [6] is to identify, address and eliminate software risks before they become threads to software operation. Evidence shows that most faults in software applications can be found in only a few of a system's components (*Moller and Paulish* [7]; *Kaaniche and Kanoun* [8]; *Ohlsson and Alberg* [9]; *Fenton and Ohlsson* [10]). *Pressman* [11] describes that experience indicates 80% of all the potential for a software project failure can be account for by only 20% of the identified risks. Choosing an optimal strategy to rank risks and identify this 20% will have a deep effect on expenses reduction and product's functionality and as mentioned by *Emam et al.* [12] it allows organizations to take early mitigating actions to detect the defects of high risk components.

A subset of the software risks is related to the poor quality of the solution and probability of the operation failure of its components. In [13] *Khoshgoftaar et al.* discuss that finding the problematic components which are more probable to fail can guide the project team to plan an optimal development which minimizes this probability and its effects. Also *Larman* [14] mentions in his book that risky components should be identified to be developed and tested in the early stages of the development to minimize risks of the project. In this study the word *Risk* is used to refer to this special subset of the software risks which represent the probability and damage of the failure of the components of the software product.

As mentioned in [15] by Cortellessa, software risk can be quantified as a combination of the following two metrics:

- Risk Probability
- Risk Damage

Finding a technique to identify these two metrics for all the components of a software project can be extremely useful in having a better overall picture of the software project. These findings can guide the development and project test planning of the software project.

There are different ways that can be used to classify components of a software product based on their risk. these techniques can be classified in three main groups:

- Analysts Estimation
- Architectural Analysis
- Source-Based Analysis

In chapter 2 each one of these groups of techniques are described and the details, advantages and disadvantages of some approaches of each group are presented and discussed.

1.2.2 Test Optimization

Software test is a critical step in a software development process. It is estimated that software testing can take up to 50% of the total development cost according to *Beizer* [16]. Also as mentioned in Section 1.1 another estimation by *Tassey* [3] shows that in US every year $20 billion could be saved if a better test could be done before the final release of new software products; therefore any technique that can help the process of software testing can have a huge effect on the success of the software project.

Risk management is one of the many techniques that can help the software test and quality assurance which focuses on reduction and prevention of *risks* in software projects. As mentioned earlier, risk refers to the probability of failure of a component and its damage. Identifying risky components which are more likely to fail can guide the management team to plan the development and testing of the software product in an optimized way.

One of the many techniques that can be used to guide the test phase of software development are source-based risk analysis techniques. Source-based risk techniques are used to classify the components based on metrics which are extracted from the source-code of each component. There are a number of advantages in use of a source-based technique for risk assessment. The most important one is the fact that in source-based analysis techniques some specific features of the technologies or programming languages can be taken into consideration. As an example the number of statements for a specific functionality can be different between different programming languages; by using the number of statements as a metric to calculate the risk the effect of used technology will be taken into account. Thus it can be seen that the source-based risk analysis can contain some parts of the technology-based risks as well. In this study in the first step area of source-based risk assessment for test optimization will be explored. In the next step this approach will be extended to other types of risk assessment techniques and a more general and precise approach will be designed. These proposed approaches will be presented in chapter 3 of this thesis.

1.2.3 Automatic bug-fix

Risk assessment can help the development team to plan the test phase of the software project and find the most damaging defects of the developing application; however fixing these defects is still a time consuming task which has to be done by developers. One of the approaches which has recently been studied and can have a promising future to make the task of debugging automatic is the genetic programming bug-fix technique. In these approaches genetic programming is used to evolve a buggy code into a bug-free code by making modifications to the structure of the code using a set of negative and positive test cases[1] to measure the fitness of each individual. As for the last part of this study, the focus is on the usage of genetic programming for automatic bug-fix. The details of the current approaches are studied and a new suggestions are made to improve their performance.

1.3 Scope and Goal

The main goal of this research is to develop a precise model and tool to facilitate the process of software reliability and risk analysis and provide the development team with practical approach of dealing with the problem of software reliability. To achieve this goal,

[1] *Positive test cases* are those which cause the application to behave as it should, and *negative test cases* are those which result in a fault.

in the first step different approaches of component classification based on risk analysis will be explored and advantages and disadvantages of each approach will be identified. The knowledge which is acquired from this study will be used to create a proposed approach of *software component classification for test optimization*.

Following this study on *risk-based test optimization* the second goal will be set to increase the precision of the proposed approach and extend it to make it more general and usable in all stages of software development process. The focus will be on finding an approach which takes advantage of different aspects of a software project and combine them to create a complete risk model which can be used to plan the development as well as test of a software product.

As for the last step of this study, the application of genetic programming for automatic bug-fix will be explored. The goal is to define an approach which is specialized to fix a set of common logical defects in the software components. A set of rules will be defined and a tool will be designed to evolve a C-like code and execute an automatic debug.

1.4 Contributions

As mentioned earlier, the purpose of this study is to facilitate the process of software quality and reliability assurance using risk analysis techniques and automatic debugging tools. The contributions of this work can be summarized as follows:

- Providing a more accurate risk model compared to the existing models which can be used to identify the risky components of the software under development with a higher precision.

- Providing useful and applicable approach to map the results of risk analysis to software development activities.
- Defining the new model in a way that can be programmed and automated to be used as a tool.
- Increasing the flexibility and usability of Genetic Programming Bug-Fix approaches.
- Applying Genetic Programming Bug-Fix operations inside statements rather than looking at statements as atomic elements of the code.

In the following chapters, the proposed techniques will be presented and it will be explained how the new approaches will satisfy these claims.

1.5 Thesis Outline

The rest of the thesis is organized as follows. In chapter 2 a review of the current studies in the area of software risk analysis and automatic bug-fix is presented. In chapter 3 proposed approaches for risk-based text optimization, component classification and genetic programming bug-fix are described and the advantages of proposed techniques over the existing ones are explained. Chapter 4 is dedicated to present the results of applying proposed approaches to a set of sample projects and in the last chapter the results of this research are summarized and the plan for further studies are described.

Chapter 2

Related Works

In this chapter the basic concepts of Software Risk analysis will be introduced and different proposed techniques that can be used to generate a risk model will be discussed. Also the advantages and disadvantages of these techniques will be explained and a basis for the next chapter will be set to describe the proposed approaches.

2.1 Risk Analysis Techniques

As mentioned in section 1 software risk can be quantified as a combination of the following two metrics:

- Risk Probability
- Risk Damage

The purpose of a risk model is to classify different components of a software product in terms of their *Relative Risk Probability* and *Relative Risk Damage*; in other words these two metrics will be used to compare different components of a software product and discover the most risky or damaging components. Identifying these components can help the project managers to plan the development of the software product in the most optimized way possible.

Different approaches have been proposed to measure these two metrics. Some of them are based on *business owners and developers opinions* about the software under development. In these approaches developers estimate the riskiness[1] of components based on their knowledge and experience. Component failure damage is estimated by developers and business owners collaboration. Some of the important factors that affect the result of this approach are: developers and business owners' experience in the area of the software under development, precise insight of the problem domain, etc.

Some other approaches are based on *formal and computable techniques* which are possible to be programmed and automated. This means that risk analysis can be done by means of an analyser application. These techniques can be categorized in two groups: The first group are those techniques that can process *the architectural design and modeling artifacts* of a component (e.g. *Statecharts* or *Sequence Diagrams*). They calculate different metrics such as *Static* or *Dynamic Complexity* to generate the *Risk Model* of each component and estimate its risk and reliability. Some samples of these approaches are discussed in the research works performed by *Bass et al.* [17], *Cheung et al.* [1], *Cortellessa et al.* [15], *Popstojanova* [18] and *Yacoub et al.* [19]. The second group are those techniques that process *the source-code of a component* and estimate the risk metrics based on the code specifications such as the number of conditional statements, function calls, etc. These values will be used to determine the risk factor of components and produce the the Risk Model of the software product. Some studies in this area are

[1]we use the term *riskiness* to refer to relative probability of failure of a component in relation to other components of the same software product. This value doesn't identify the absolute probability of failure.

done by *Deursen et al.* [20], *Hosseingholizadeh* [21] and *Wong et al.* [22]. This Risk Model can be a very useful tool to manage the debugging and testing strategies of a development process.

Each one of these techniques have some advantages and disadvantages. The *owners and analysts opinion* is a very important factor to determine the risky components, because risk discovery is a heuristic process and no two projects are exactly alike. However this type of analysis cannot be sufficient because many of the risks are not determinable by analysts due to lack of experience, unexplored areas of new technologies that are used in every different project, implementation details which are not clear before the design and coding phases begin, weak points of development team in the area of the new technologies, etc. In addition to these, there is always the human fault factor which causes some risks to remain undetermined.

Architectural analysis techniques are not enough either because an application with a simple architecture can have a very complicated internal logic in the body of the functions and procedures. These risks can only be determined by code analysis techniques.

On the other hand, just using *Code-Based analysis techniques* will not generate a precise Risk Model because the internal structure of the functions and procedures of an application can be simple, but the relation between its components can be very complicated.

Considering the above and different phases of a development process (design and implementation of each subsystem[2]) which take place before the test, it can be seen that a risk analysis approach can produce a reliable model only when it uses the information obtained from the combination of all the analysis, design (architecture) and implementa-

[2]A subsystem is a set of closely related components and classes which is a part of a bigger system and is not intended to be used separately

tion (code) phases. In section 3.2 a new approach will be proposed in which the owners and analysers' estimation is considered as one of the effective factors and it is combined with the values that are obtained from Architectural and Source-Code analysis. This will result in a compound model that takes all three aspects of a risk analysis into consideration. The idea of combining these three aspects can be summarized in the formula 2.1.

$$RiskMetric = f(AnalystsEstimation, ArchitecturalRisk, SourceBasedRisk) \quad (2.1)$$

In the following sections of this chapter we go into some details of Architectural and Source-Based Risk Analysis approaches and their usages.

2.2 Architectural Risk Analysis

Architecture of a software product has a deep effect on its reliability. Proper multi-tier architecture, proper dependencies between components, right abstraction and encapsulation of objects, etc. are some of the architectural factors that effect the reliability. In [23] we used these techniques in a specific type of applications (Wireless Sensor Networks) to show that having a properly structured architecture can facilitate the development and increase the reliability of the product. Many of the reliability and risk analysis techniques have focused on the architectural specifications of a software product.

Cheung et al. [1] discovered the reliability factor of each component by finding the state transition diagrams of the components of a software product and using them as Markov Models to calculate the probability that a component ends up in a failure state. A sample of such a state diagram is shown in Figure 2.1. In this figure the failure states are shown in grey.

In another approach *Popstojanova* [18] calculated the *Cyclomatic Complexity*[3] of the state transition diagrams of the components of the software under analysis.

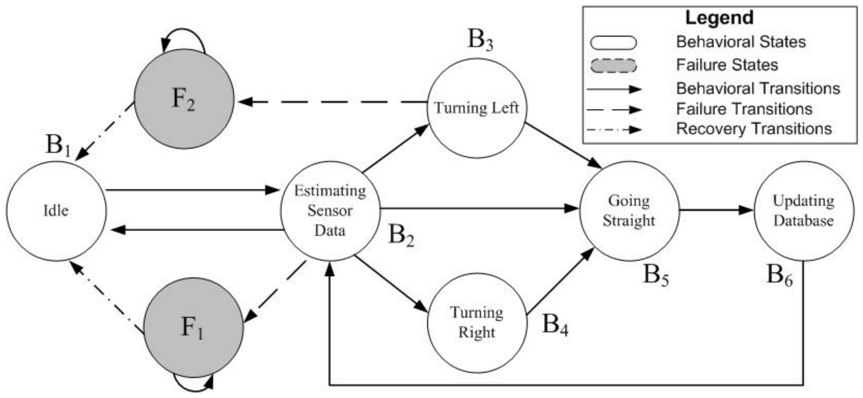

Figure 2.1: A sample state transition diagram of a robot controller discussed by *Cheung et al.* in [1]

Another approach is proposed in [15] by *Cortellessa et al.* in which the focus is on the *Performance Risk Analysis* by assigning a demand vector to each action/interaction in the sequence diagrams of different system scenarios and using the resulted execution model to calculate the service demands (work units for CPUs and KB[4] for the disks and network) and service time of the specific used hardware. These values are later used to create the System Execution Model based on the workload parameters and estimate the probability of failure as a violation of the performance objectives.

The purpose on all architectural risk analysis techniques is to determine the complication of the dependencies between different elements of a software product. Having unnecessary dependencies between components, wrong abstraction and encapsulation,

[3]Cyclomatic Complexity is a metric which was first proposed by McCabe [24] to find the number of linearly independent paths of execution in a program based on its control flow graph and determine its complexity and later on was adapted to other types of software architectural graphs.

[4]Kilobytes

etc. are some of the reasons that could increase the complexity of the components and make them more vulnerable to reliability problems and increasing the architectural risk factor.

2.3 Source-Based Risk Analysis

Source-based risk analysis techniques analyse the source-code of an implemented software product and detect its risky elements and components. A source-based technique can be implemented as an automatic tool in an IDE[5] to be used by developers and classify the implemented components.

In [22] *Wong et al.* based their approach on analysing the source-code by assigning a weight to members of a set of code-related metrics (such as number of function calls, variable definitions, etc.). These metrics are calculated (with consideration of their corresponding weight) then the result of their multiplication or addition (based on different schemas proposed in [22]) is used as the risk factor of each element of the software i.e. block, function or component. In [20] *Deursen and Kuipers* proposed an approach to create a risk model based on the information retrieved from the interviews with stakeholders, and use a source-based technique to verify/modify the risk model. Many of the risk analysis and classification techniques are based on the idea of *Cyclomatic Complexity* introduced by *McCabe* [24]. Cyclomatic Complexity measures the complexity of a source code and can be used to identify risky (complex) components.

[5]Integrated Development Environment

2.4 Software Test Optimization

Different approaches have been proposed to assist the software test process: *Lei et al.* [25] and *Carlos et al.* [26] proposed different techniques for test-case generation, *Nguyen et al.* [27] and *Forrest et al.* [28] focused on automatic bug-fix techniques, *Khoshgoftaar et al.* [13] proposed a new approach for components classification using genetic programming-based decision trees, etc.

As mentioned in the previous section, a part of risk assessment techniques are based on source-code analysis of the target software product. Some studies has been done on this topic such as researches by *Wong et al.* [22], *Hosseingholizadeh* [21] and *Deursen et al.* [20]. These techniques can be used to identify the risky components and the development team can use this information to plan the software testing phase and make it possible to address maximum number of defects in a limited time.

The problem with these techniques is that they only identify the risky elements of the software without any observation over the amount of the effort which is required to test them. Also most of these approaches have a linear relation with the number of statements inside a component; thus huge components with huge number of functions and elements are associated with higher risks. These components can take a long time to be tested and fixed, but due to their high risk they get a higher priority over the small and testable components with fewer but more critical operations. As it will be presented in section 3.1 we will show how our risk model takes the factor of effort of fixing an issue into consideration and increases the performance of the testing phase of software development process.

2.5 Automatic Bug-Fix

To ensure the quality of the final product the debugging process should be organized and closely managed in all the iterations of the software development and subsystem release. But still considering the limited human resource and human error factor, there is a limitation to the quality improvement that can be applied to the product in a limited period of time. One of the approaches that can assist the development by applying a precise logical debugging is genetic programming automatic bug-fix.

Genetic Programming has a very special characteristic which is the ability to generate valid code. This feature makes it a very good candidate for automatic software bug-fix. The basic idea is to use a GP system to evolve a buggy code into a bug-free code. Different approaches have been proposed to achieve this goal in the recent years by *Forrest et al.* [28], *Weimer et al.* [29], *Nguyen et al.* [27], *Arcuri et al.* [30] [31]. To be able to evolve the individuals, a set of positive and negative test cases are prepared. These test cases are used to measure the fitness of individuals. The GP system tries to apply changes to the original code and find a candidate which passes all the test cases.

One of the challenges in using GP systems in software repair is the huge search space of different codes that can be generated by evolving the original code. In [27] *Nguyen et al.* mentions this makes it very hard for the GP to converge a source code to an evolved code which retains the functionality of the original code as well as fixing its bug(s). Crossover operator in GP systems usually applies big structural changes in the individuals and this can have a negative effect for the bug fixing process. In the context of software repair big changes disrupt the system from converging toward a bug-free code without disrupting the original functionality of the source. To prevent this *Forrent et al.* [28] and *Nguyen et al.* [27] use a modified crossover which applies the operation around a part of the code which seems to be the problematic part.

There are different ways to identify pieces of code which are candidate of being the problematic blocks and statements. In [27] a weight is assigned to each statement which is a part of a execution path[6] that results in a bug. This weight is used to identify the problematic statements. In [28] *Forrent et al.* assign 3 types of probabilities to the statements; first probability is assigned to those which are only a part of execution path of a negative test case, second probability is assigned to those which are a part of the execution path of both negative and positive test-cases, and the last one is applied to the rest of the statements. These probabilities are multiplied by the probability of a GP operator and the result is used to apply the crossover and mutation operators to the individuals.

Using these weights the genetic programming system can explore the search space by modifying the problematic statements and thus it shrinks the search space of the genetic programming system and increases the chance of converging toward a bug-free code; however the drawback of this limited search space is excluding some very good fixes which need bigger code manipulation. To overcome this problem, the proposed approach in [27] applies two types of crossover; the first one is the limited crossover which swaps only those statements which are in the execution path of negative test cases, and the second one is a conventional one point crossover which swaps two statements from two individuals regardless of their participation in the execution path of test cases. These two types of cross over are applied to the population with different probabilities.

In both approaches in [28] and [27] the authors use existing statements in the application to create a code-bank and they use this bank to apply the mutation operation to

[6]An *execution path* is a sequence of statements in one code which are executed in a run. This sequence can be automatically logged by the interpretor of the code.

individuals by swapping a random statement from the buggy module with a statement in the code-bank; thus a mutation is simply swapping a statement with another statement from the source-code of the same application. Also crossover is limited to swapping one or a group of statements of the individuals with one another. During the process of evolution, non-working programs with syntactical problems are eliminated in the selection phase.

Procedure 1 Microsoft Zune buggy code

```
void zunebug(int days) {
   int year = 1980;
   while (days > 365) {
      if (isLeapYear(year)){
         if (days > 366) {
            days -= 366;
            year += 1;
         }
         else {
         }
      }
      else {
         days -= 365;
         year += 1;
      }
   }
   printf("current year is %d\n", year);
}
```

Both approaches proposed in [28] and [27] have shown that they are able to fix different bugs in some sample applications. In sample Procedure 1 a famous buggy code which was reported in Microsoft Zune players is shown which falls into an infinite loop when the input **days** is the last day of a leap year. This buggy code was fixed by the GP auto-repair system presented in [28] with the modification presented in Procedure 2.

Procedure 2 Microsoft Zune bug fixed by GP technique proposed in [28]

```
void zunebug_repair(int days) {
   int year = 1980;
   while (days > 365) {
      if (isLeapYear(year)){
         if (days > 366) {
            // days -= 366; // repair deletes
            year += 1;
         }
         else {
         }
         days -= 366; // repair inserts
      } else {
         days -= 365;
         year += 1;
      }
   }
   printf("current year is %d\n", year);
}
```

Chapter 3

Methodology

In this chapter a description of this study over the topic of Software Risk Analysis will be provided and the details of the contributions to this field which are published in [32] and [21] will be presented. The main concern of this research is to develop some analytical techniques which can be used in real world projects in form of automatic tools. To accomplish this, after studying different approaches of Software Risk Analysis, Defect Measurement, Bug Localization and Classification, Software Design Techniques and Automatic Bug Repair techniques the focus is turned toward studies on source-code analysis and a new source-based risk analysis technique is developed. This technique can be used to assist the process of software test and optimize the use of resources (this approach is represented in section 3.1). Continuing this research, the source-based technique is extended in a more general form which could be used to analyse the structure of a software from different points of view (source code, architectural, etc.) and generate a more precise and reliable risk model of the software components (section 3.2). Moving on to the next level, intelligent automatic bug-fix techniques are studied and a new technique based on Genetic Programming is developed which can evolve a defective code into a bug-free code. In the following sections these approaches and the results of using them in software projects are described.

3.1 Risk-Based Test Optimization

In this section our proposed approach which is published and presented in [21] will be described. This approach results in a model that can be used to plan an optimal test for the software project. A code analyser tool is designed based on this proposed technique. This tool is used to apply this technique to a sample project. The results of this experiment are presented in the next chapter.

There are two phases in our proposed technique:

- Components Classification
- Source-Based Risk Analysis

The following sections will describe these two phases.

3.1.1 Components Classification Based on Their Importance

In the first step business owners and software developers should classify the components of the software based on their importance to the business and the functionality of the software product. In order to do that, since business owners usually don't have the required technical knowledge to help the component classification process, they will only assign a number between 0 and 100 to each use-case based on their importance to the business (100 would stand for the most important use-case). Considering the operation of each component in each use-case software developers can use this metric to help them determine the importance of each component and classify them. Developers use this metric to classify the components in 4 classes of *Negligible, Marginal, Critical* and *Catastrophic* which represent the class of damage that can be caused by failure of each component.

3.1.2 Source-Based Risk

After classifying the components, a source-based approach is used to determine the riskiness of each component and its elements. As mentioned in the previous chapter, different metrics can be used to determine the riskiness of components. For example number of lines of code, number of data access, cyclomatic complexity (based on control flow) are some of different approaches which are developed and used by *McCabe* [24], *Popstojanova* [18], etc. Our proposed approach is based on the idea presented by *Wong et al.* in [22] which analyses the source-code to determine the risk of an element of the application. To accomplish a better result, this idea is improved by adding more structural observations to the analysis process which results in a better estimation of the risk of the component-under-analysis. In [22] *Static Risk Model* based on *summation* scheme is described as following:

$$V * \alpha + F * \beta + D * \gamma + C * \epsilon + P * \rho \tag{3.1}$$

In this formula V, F, D, C and P are metrics extracted from the code. V stands for number of variable definitions, F number of function calls, D number of decisions, C number of c-uses[1] and P number of p-uses[2]. $\alpha, \beta, \gamma, \epsilon$, and ρ are the weighting factors which are used to give either more or less emphasis to the metric components.

[1]c-use(computational use) is defined for each block of code as the number of variable usages in the right hand side of each assignment statement, plus the number of variable usages in output commands.

[2]p-use(predicate use) is defined for each block of code as the total number of variable usages in conditional statements.

Our proposed technique is an improved version of the former logic. This technique can be explained using the following two sample procedures shown as *Procedure 3* and *Procedure 4*.

Using the formula 3.1 these two procedures will have the same value for risk; however

Procedure 3 - A procedure with one statement in the *if* body

```
Procedure foo()
{
   if (condition 1)
   {
      statement 1;
   }
   statement 2;
   statement 3;
   statement 4;
}
```

Procedure 4 - A procedure with 3 statements in the *if* body

```
Procedure bar()
{
   if (condition 1)
   {
      statement 1;
      statement 2;
      statement 3;
   }
   statement 4;
}
```

by further analysis it can be seen that *Procedure 4* is more risky than *Procedure 3*. The reason for this is that if in *Procedure 3* as an example the *condition 1* fails to operate properly, more statements will be executed which are not supposed to be executed; thus it can be said that *condition 1* in *Procedure 4* is more damaging than *condition 1* in *Procedure 3*. This concept can be interpreted by saying that the riskiness of *condition 1* is added to the riskiness of *statements 1, 2* and *3* in *Procedure 4* and made them more risky (As an example, it can be said that the riskiness of *statement 2* in *Procedure 4* is based on two factors, first the risk of failure of *statement 2* itself and second the risk of failure of proper execution of *condition 1*). With this approach the effect of the risk of *condition 1* can be interpreted as an increase in the riskiness of its corresponding block of statements.

The former point is valid for all the statements in conditional/loop blocks (such as *for*, *while*, etc.). Having a bug in the condition of a loop will effect its whole body by the measure of the number of its wrong executions; thus the risk factor affects the body of the loop in a more severe form. It is more complicated to measure this effect in loop statements because in some cases the number of executions of their body is not determinable prior to the application's execution. To make this measurement simpler we consider a constant weight to be assigned to all the loops which should be determined by the developers and project analysts.

Considering the former points, the formula 3.1 in our approach is changed into the following equation:

$$BR(n) = V' * \alpha + F' * \beta + D' * \gamma + C' * \epsilon + P' * \rho + \sum BR \tag{3.2}$$

In this equation $BR(n)$[3] stands for riskiness of block n, and $\sum BR$ stands for the sum of all *BR*s of the internal blocks of block n. If we represent all the metrics *V, F, D, C* and P with X, all X's in (3.2) are defined as follows:

$$X' = X * BRF(n) \tag{3.3}$$

In this formula $BRF(n)$ represents the *Block Risk Factor* of the block n. $BRF(n)$ in our proposed approach is defined as following: In the first level of each procedure (or

[3]Block Risk

function) BRF is 1; by entering each if statement block, BRF would be equal to the BRF of if statement's parent block incremented by 1. BRF of each loop block is equal to BRF of loop's containing block plus 5 (this is our proposed value and our approach can be applied with other values for different projects based on their implementation specifications). To explain this technique the example shown in *Procedure 5* is provided.

Procedure 5 - A sample procedure

```
Procedure example()
{
   int a, b;
   int max;
   cin >> a >> b;
   if (a > b)
   {
      max = a;
      cout << max;
      Proc2();
      for (int i = 1; i <= max; i++)
         Proc3();
   }
   else
   {
      cout << b;
   }
   cout << a * b;
}
```

In *Procedure 5* for the outer most procedure, BRF is 1. For *if* and *else* blocks BRF is equal to 2 (BRF of the parent block + 1) and for the level 3 block which is inside the *for loop* BRF is equal to 7 (BRF of the containing block + 5). In the inner most block (*for loop*) which is considered as block number 4 there are 2 p-uses ($i <= max$), a c-use (*i++*), a decision and a function call (*Proc3()*). By assuming that all the weights are 1:

$$\begin{aligned} BR(4) &= V * BRF(4) + F * BRF(4) \\ &+ D * BRF(4) + C * BRF(4) + P * BRF(4) + \sum BR \\ &= 0 * 7 + 1 * 7 + 1 * 7 + 1 * 7 + 2 * 7 + 0 \end{aligned}$$

in the *if* block there is a variable definition (since *int i* is executed only once it is considered outside the *for loop* body), 2 c-uses (*cout* << *max* and *max* = *a*), a function call (*Proc2()*) and a nested block (BR(4)):

$$\begin{aligned} BR(3) &= V * BRF(3) + F * BRF(3) \\ &+ D * BRF(3) + C * BRF(3) + P * BRF(3) + \sum BR \\ &= 1 * 2 + 1 * 2 + 0 * 2 + 2 * 2 + 0 * 2 + BR(4) \end{aligned}$$

With the same approach, BR(2) for the *else* block and BR(1) for the procedure's block are calculated as follows:

$$\begin{aligned} BR(2) &= 0 * 2 + 0 * 2 + 0 * 2 + 1 * 2 + 0 * 2 + 0 \\ BR(1) &= 3 * 1 + 0 * 1 + 1 * 2 + 2 * 1 + 2 * 1 + BR(2) + BR(3) \end{aligned}$$

Finally the risk of the outer most block which is the example procedure (*BR(1)*) is equal to 54.

Also another new metric called *Function Risk Density* is defined in our approach as following:

$$FRD = \frac{FR}{LOC} \tag{3.4}$$

In equation 3.4 *FRD* stands for *Function Risk Density* that is obtained by calculating *BR* of a function (which is called *FR*[4]) and divide it by *LOC* which stands for the *Lines of Code* of the code under analysis. Higher values of *FRD* show that the function has more nested statements; higher values also show that regardless of the size of the function, each single statement in the function has a high risk. Giving higher test-priority to the functions with higher *FRD* will result in eliminating more risks by testing each line of code. The risk of a component can be calculated by adding the risks of all the functions in it:

$$CR = \sum FR(i) \tag{3.5}$$

This metric is a relative metric which can only be interpreted in relation to other components' risks and it represents the overall relative risk of a component. Also *Component Risk Density* can be calculated with the following formula:

$$CRD = \frac{\sum FRD(i)}{N} \tag{3.6}$$

In this equation N stands for the number of functions of the component. This value can be used as the estimated average risk of statements in the component. In another word if a component has a higher *CRD*, by testing each of its statements (or functions) a higher risk can be eliminated. Thus testing those components which have higher *CRD* would result in a more optimal test.

[4]Function Risk

3.1.3 Risk-Based Testing

After classification of components and obtaining *CRD*, *FRD* the test phase can be planned. In the first step, critical components based on the classification in section 3.1.1 should be chosen to be tested. The test phase should start with a component from the *Catastrophic* class which has the highest CRD. The test of each component should be conducted by starting with the functions with higher *FRD*. A threshold T will be chosen for each component which will be used to select the functions to be tested; only the functions with a *FRD* above this threshold should be tested. This is used to eliminate very simple functions from being included in the test plan. The value of this threshold should be chosen based on the *size of the project* and *available test time*. After the test of all catastrophic components, the remainder testing time can be used to continue the test on *Critical*, *Marginal* and *Negligible* components respectively. Using this approach it is possible to optimize the software test process and verify and fix the maximum possible amount of the components in a given time. A case-study of applying this approach to a sample project is provided in section 4.1.

3.2 Compound Risk Model

In this section the proposed approach of this study for software risk model is presented. This approach is published in [32]. Following the study on Source-Based Risk Analysis, in this section the goal is to increase the precision of risk analysis by considering more risk related factors in the calculations. This improvement resulted in a new approach which is called *the Compound Risk Model*. This approach is based on a combination of different risk factors which take different viewpoints of the software project into account to generate the risk model. Each one of the source-based, architectural based and analysts estimation techniques of risk model generation reflects the reliability measure of a software project from a specific angle. By using all the available techniques and combining the results a more precise model can be generated. In the following sections, first our proposed approach of generating and combining the riskiness (relative probability of failure) of all the components is described, then the proposed method of risk damage estimation and aggregation of these information will be presented which will be used to generate the risk model.

3.2.1 Riskiness of Components

In the first step the proposed approach to determine the riskiness of a component of an application will be introduced. The goal is to define a technique to compare the risk of different components of a software product. As discussed in the previous section our approach is based on a compound metric which takes all aspects of a project into consideration. First the elements of this compound metric and their calculation method will be presented, and after that the method of aggregating these elements and calculating the compound metric will be described.

3.2.2 The elements of the riskiness compound metric

To determine the relative probability of failure of each component (in relation to other components of a software product) its riskiness factor is calculated by considering the following viewpoints:

- Project Analysts and Managers' estimation about the probability of failure of each component

- Architectural Analysis of each component
- Source-Code analysis of each component

3.2.3 Component riskiness based on analysts estimation

Project managers and analysts can use their experience to determine the relative riskiness of each component. In order to do this, after each design session analysts and project managers will discuss each component of the designed subsystem and associate a number between 0 to 100 to each component which identifies its riskiness (The higher the number is, the more probable is the failure of the component). This value is shown by CR_1.

In short term: $CR_1(k) =$ *The riskiness factor of the component k which is determined by the project analysts and managers.*

3.2.4 Component riskiness based on architectural analysis

In 1976 McCabe proposed a technique to determine the complexity of an application [24]. In his method a new metric called *Cyclomatic Complexity* was introduced which determined the complexity of an element of an application by evaluating $V(G) = E - N + 2$ using the element's *Control Flow Diagram.* In this equation $V(G)$, E and N stand for *Cyclomatic Complexity*, *number of Edges* and *number of Nodes* respectively. In [18] *Popstojanova* extended this definition of complexity to the software architecture level. In his proposed approach a *Statechart* is designed for each component which is used instead of the *Control Flow Diagram* to make it possible to calculate the complexity in the Architectural level. This complexity has been used as the architectural risk factor of a component. In our study the architectural riskiness for the *Compound Risk Model* is calculated based on the idea of [18].

In the approached proposed in [18], in order to determine the Architectural Risk, the statechart of each component is designed based on each use case; then cyclomatic complexity is calculated based on this diagram (using the number of edges and nodes). In our study each component is considered from a general point of view, thus separate statecharts will not be designed for different use-cases; instead the proposed approach is to design only one statechart which contains one default or idle state and all the other possible states of a component based on all use-cases. The Complexity Factor of component k will be calculated using the following equation:

$$CR_2(k) = E(k) - N(k) + 2 \tag{3.7}$$

In this equation $E(k)$ and $N(k)$ represent the number of edges and nodes for the statechart of component k, and $CR_2(k)$ stands for the Cyclomatic Complexity of component k which represents its riskiness and subsequently its relative failure probability factor. In section 3.2.6 this value will be combined with other metrics and normalized to produce a compound metric that can be used to determine the riskiness of a component.

3.2.5 Component riskiness based on source-code analysis

In this section an approach to determine the riskiness of a component by source-code analysis will be proposed. As mentioned earlier there are different approaches which use different factors to calculate the source-code's riskiness: number of lines of code, number of data accesses, cyclomatic complexity (based on control flow), etc. For the purpose of our study, the same approach which was proposed in section 3.1.2 is used. Using this approach, a better estimation of the riskiness of blocks of code can be obtained. This estimation takes the structural specifications of the source-code into account as well as types of statements. By considering formula 3.2 the source-based riskiness will be calculated. First all *BR(i)*s of all the blocks of code and procedures in a component will be obtained. Then the riskiness of a component can be obtained by calculating the sum of the riskiness of all the procedures inside it:

$$CR_3(k) = \sum BR(i) \tag{3.8}$$

Again it should be mentioned that this metric is a relative metric which can only be interpreted in relation to the riskiness of other components. Also another metric is defined which is called CAR and stands for Component Average Risk:

$$CAR_3(k) = \frac{\sum BR(i)}{N} \tag{3.9}$$

CAR_3 stands for the average of riskiness of all the methods inside the component k; N is the number of methods inside a component (The index 3 in CR_3 and CAR_3 is used to distinguish these metrics from other metrics which were introduced in the previous sections). If a component has a relatively bigger CR_3 then it can be said that this component is riskier than others; having a component with a relatively bigger CAR_3 means that the density of risk in this component is higher.

3.2.6 Combining the metrics

In this section the purpose is to combine the acquired metrics from the previous sections to generate the new compound metric. CR_1, CR_2 and CR_3 will be combined and a new metric CR will be created which will determine the Component's Risk. By comparing CRs of different components in a project, one can identify the most risky components.

In the first step the values of CR_is should be normalized using the following formula:

$$NCR_i(k) = \frac{CR_i(k)}{\sum_k CR_i(k)} \tag{3.10}$$

In (3.10) $NCR_i(k)$ stands for *the normalized riskiness of component k*. The parameter i identifies each one of the 3 metrics which where calculated in the last three sections. $\sum_k CR_i(k)$ represents the sum of CR_is of all the components. Using (3.10) we will normalize all the calculated values for all three metrics and the result will be three values of $NCR_1(k) \sim NCR_3(k)$ for each component of the developing application. NCR_1 stands for normalized riskiness estimated by *project managers and analysts*, NCR_2 stands for normalized riskiness calculated by *Architectural Analysis* and NCR_3 stands for normalized riskiness calculated by *Source-Based Analysis*. After normalization, the compound metric *CR* (Component Risk) is calculated using the following formula:

$$CR(k) = \theta * NCR_1(k) + \omega * NCR_2(k) + \sigma * NCR_3(k) \tag{3.11}$$

In this formula $CR(k)$ stands for the riskiness of component k, $NCR_i(k)$ stands for the normalized riskiness of the three mentioned approaches and θ, ω and σ are the weighting factors which are used to give more or less emphasis to each of the metric elements. Considering that this formula uses all the aspect of a software project, it can be seen that the classification of components based on CR(k) is much less faulty than other methods. This method can be used in any stage of a software development. If the risk analysis is performed at the early stages of development (before any implementation) CR(k) can be calculated by putting 0 as the value of NCR_3, thus CR(k) can be calculated without any implementation. Considering that the value of CR for one component is a relative value and it is eventually used by being compared to CR of other components, the obtained values can be used without any change in the original algorithm.

3.2.7 Risk Damage Analysis

In this section our proposed approach to determine the damage that each risk can cause in a software product will be described. Using this proposed approach, components can be classified based on their potential damage. Components' failure damage will be determined based on the project's *Analysts and Managers estimation* of the damage of each component and *Architectural Analysis* of the software structure.

3.2.8 Failure Damage Based on Analysts Estimation

The logic of the application is a very important factor in a component failure damage. The damage caused by a failure is defined by the damage of the failed tasks of the application to the business operation; thus it can be seen that analysts and managers estimation of the wrong execution of a component is very important. The most important resources that analysts can use to clarify the business logic of the developing application are the business owners; but the problem of this strategy is that usually owners don't have enough understanding of the internal structure of a software and a middle step should be taken to map owners' view of the system to the actual structure of the application. In the context of Risk Analysis, the operations which are important to the business and their responsible components should be identified. The proposed method of this study for this mapping is as follows:

In the early sessions of analysing the new software project which take place between business owners, analysis and developers, use-cases of the developing application should be discovered and identified. After this identification business owners should assign a value between 0 to 100 to each use-case which determines the importance of that use-case. A use-case with the value of 100 would be the most important use-case. Considering this value, analysts and developers can discover the participating components in each use-case and identify the most important components of the software. They would assign a value between 0 and 100 to each component based on their knowledge about the degree of participation of components in the use-cases. Knowing that it is not possible to involve business owners in the structural design of the application, this approach helps the developers and analysts to map owners' view to their design. In this approach the importance of each component will be shown by **imp(k)** which stands for the importance of component k extracted by analysts and developers from importance of use-cases. Since $imp(k)$ is determined in the first steps of the development, as the application is being developed its architecture will be modified, thus developers might need to re-assign or modify the values of $imp(k)$ for different components.

The new metric *Internal Operation Failure Damage (IOFD) of component k* is defined as follows:

$$IOFD(k) = \frac{imp(k)}{100} \tag{3.12}$$

In this equation $IOFD$ would have a value between 0 and 1; higher values represent more damaging components.

3.2.9 Architectural Damage and Metric Aggregation

In this section a new factor regarding the failure damage of software components is introduced. In this proposed method the failure damage depends on the *Dependencies* between components. To clarify this first the *Depth* concept should be introduced. Having the package diagram of an application, the *Depth of a package* is defined by assigning numbers to the packages of the application starting from 1 which is assigned to the upper (outer) most package and moving down assigning 2, 3, ... to the packages in the lower tiers, and the *Depth of a component* is defined as the number assigned to its containing package (Figure 3.1).

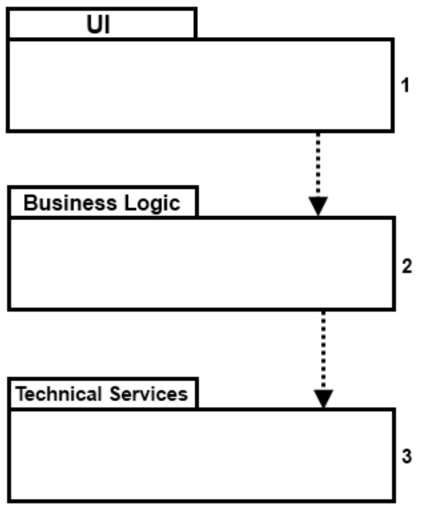

Figure 3.1: Depth of each component is determined by the depth its container package

Since in a package diagram the functions of the higher components depend on the correct execution of the lower components, therefore in case of a failure if a component has a higher value of depth it will cause more damage. As an example if there are multiple components in the second tier of the package diagram (depth : 2) which are all dependent on one component in the 3rd tier, a failure in the operation of the lower component can potentially cause the failure of operation of all the higher components. This dependency can be between components of the same package as well. Dobrica mentions this point in [33] that high interaction between unrelated scenarios indicate a poor separation of functionality which is correlated with the number of defects in the final product.

Considering the former points it can be said that those components which have more dependent elements cause more damage. Considering this dependency factor, the failure damage of a component is defined as following:

$$CFD(k) = IOFD(k) + \tau * DCFD(k); \tag{3.13}$$

In this equation $CFD(k)$ stands for *Component Fail Damage*, $IOFD(k)$ stands for *Internal Operation Fail Damage* and $DCFD(k)$ stands for *Dependent Components Fail Damage* of component k. τ is the weighting factor. Using equation (3.13) the effect of the architectural design of the components of an application is also considered in the failure-damage estimation. $DCFD$ of a component is obtained by calculating the sum of CFDs of all its dependent components. This process should start from the upper-most tier (depth 1). $DCFD$ for the components in the upper-most tier which have no dependent elements is 0, thus in this case all CFDs are equal to $IOFD$s which are extracted from analysers and owners estimation of the failure damage and have a value between 0 and 1. In this step 0 means *not damaging* and 1 means *very damaging.* After the first tier CFDs of the second tier can be calculated by adding $IOFD$ of each component with sum of CFDs of its dependent components ($DCFD$).

By comparing the value of CFD for different components the most damaging components can be identified.

3.2.10 Compound Risk Table

In this section the values of CR and CFD will be used to create a table called the *Compound Risk Table.* The rows of this table can be sorted based on the Failure Damage and Riskiness (probability of failure) to determine the most harmful and risky components.

This table can be used to identify the most damaging and risky components and as a guide for development scheduling and software test and debug management.

The value of *CR* is obtained by adding up three NCR_is and since each NCR_i has a value between 0 and 1, therefore the value of *CR* is a number between 0 and 3. According to this, the risky components are those with a *CR* closer to 3 than other components.

Since the number of tiers of a software product is different among different applications, the boundaries of *CFD* can not be pre-identified. If different components have so many dependent elements in the higher tiers the value of *DCFD* will increase very fast and as a result the value of CFD can raise up to big numbers. The value of *CFD* will be used to categorize the components of an application in four classes:

- Negligible
- Marginal
- Critical
- Catastrophic

To find the membership of different components in these classes the highest value of *CFD* will be identified and divided into 4 equal number ranges corresponding to the four classes of damage (the range with lowest *CFD*s identify the category *Negligible* and higher *CFD*s identify the *Marginal, Critical* and *Catastrophic* classes respectively). The class of the components can be determined by identifying the membership of each of the components to these groups. Having *CR*, *CFD* and *Damage Class* of all the components, the *Compound risk table* will be created as shown in table 3.1.

	CR	CFD	Damage Class
Component 1	CR(1)	CFD(1)	[DamageClass]
Component 2	CR(2)	CFD(2)	[DamageClass]
...	...	...	...
Component N	CR(N)	CFD(N)	[DamageClass]

Table 3.1: Example of a Compound Risk Table

3.3 Genetic Programming Bug-Fix

In this section the proposed approach of this study for automatic software repair will be described. The basic idea of this approach is to specialize the operation of the GP system to apply more intelligent modifications which lead the process toward the optimal fix of the buggy program.

3.3.1 Genetic Programming Key Concepts

Genetic Programming is an approach inspired by *Biological Evolution* which is used to explore and find computer programs to solve a problem. This process is done by creating and manipulating a group of programs (called *individuals*) which are candidates to solve the problem. This group is called a *population*. Each candidate is evaluated by receiving a number of inputs and generating outputs which are compared to a list of desired outputs that determine the correctness of the individual. The error between the desired output and the individual's output is called *Fitness*. Each pair of input and output in the list of inputs and desired outputs is called a *Fitness Case*. The manipulation process is done by means of *Genetic Programming Operators*. There are two main operators in GP:

- Crossover
- Mutation

Crossover is an operator which works by switching two parts of two individuals (*parents*) with one another which results in two *child* individuals. *Mutation* is an operator which applies a change to one part of an individual. This change is *fail-safe* to guarantee that the resulted individual will have a valid logical structure. The resulted individuals from these two operations will be used to create a new population called the *new generation.* In addition to the former two operators there is the operator of *Reproduction* which copies a portion of the best candidates (determined by their fitnesses) of the previous generation to the new generation. The process of evolution continues until an individual is found which passes all the fitness cases.

In the next section the proposed approach of this study is described which uses a GP system to apply logical fixes to a buggy code.

3.3.2 GP Operations Scope

In the proposed GP approaches by *Forrest et al.* [28], *Nguyen et al.* [27] and *Arcuri et al.* [30] the criteria of applying GP operators are set to statement level, meaning that the GP system does not change anything inside the statements. This assumption has many benefits as well as some drawbacks. In many cases a bug in a system could be caused by simply a human error which as an example can be caused by using the + sign instead of -, or > instead of <. Since the mentioned techniques can not manipulate the contents of the statements, the proposed techniques fix these issues by creating new execution paths and eliminating faulty statements. However these changes are very complicated compared to the ideal fix which is basically a simple symbol replacement.

To fix this issue our proposed approach is to define two sets of GP operators which are *Inner Statement Level* and *Outer Statement Level* operators. The outer statement level of operators would follow the basic idea of the previously proposed techniques; but the inner statement level of operators apply changes to the statement elements by using a specific set of predefined bug types. In this sense a + can be swapped by or mutated to a -, * or /. Also conditional statements would have a set of templates to be used for genetic operations.

Crossover operator is a relatively disrupting operator and if it is not controlled it can move the population out of the domain of the solution. To handle this the crossover will always be applied in the *outer statement level* and choose one parent from a randomly chosen individual from the population and another parent from a copy of the original tree of the code to be debugged; this helps the GP system to do the search over individuals which are close to the original code in the search space. Mutation is applied to both *outer* and *inner statement levels*. Mutation in the *outer statement level* is applied to one statement of an individual which adds a randomly chosen statement from another individual after or before the selected statement. Mutation in the *inner statement level* is applied to binary and comparison operators $(+, -, *, /, >, <, >=, <=, == \text{ and } !=)$ and the constant values in the statements.

The probability of applying each GP operation effects the disruption of individuals. The proposed values in the Bug-Fix approach of this study are as follows: To reduce the amount of disruption of structure of individuals mutation (which is a minor change) should be applied with a higher probability of 0.5 compared to other operators. Crossover should be applied with the low probability of 0.2 and reproduction should be applied with the probability of 0.3 to maintain the similarity between generations.

3.3.3 Constant Repository

Many of the bugs are caused by the boundary values of the operations and statements of the programs's functions. As an example in many cases the termination condition of a *for loop* is mistakenly implemented as `(i < Length)` instead of `(i < Length + 1)`. Also in many cases the same mistake happens for initial values, length of data collections, etc. Considering the high frequency of these types of bugs, our proposed approach creates a bank of all the constant values that can be found in the *method under repair*. This bank is called *The Constant Repository*. For each integer C which is chosen to be included in the constant repository, we also include *C + 1* and *C - 1* to cover the boundary values. Whenever a mutation operator wants to choose a constant value to be replaced, it will choose a member from the constant repository (with the probability of P_c). Using this approach, the GP system would be able to modify the boundary values and cover a big portion of this common type of bugs.

3.3.4 Fitness Evaluation

The primary factor of the fitness is the number of passed test cases. If an individual returns the wrong output, falls into an infinite loop or throws a runtime exception for an input, it will be considered as a failed test case.

In [31] *Arcuri* mentions that the assumption in an automatic bug-fix application is that the buggy program is structurally close to the optimal solution. This assumption comes from the fact that the program has been written with the intention to provide a specific functionality and usually the bug is the result of a human error in some elements of the written program, or a very special set of inputs (like the Microsoft Zune bug) which need to be handled by a small modification. Thus in the context of software repair, the diversity of the population of individuals is not as contributing as other applications of GP. To handle this the fitness should be evaluated by considering the following two factors:

- number of passed test cases
- similarity to the original code

However it is possible that the optimal solution could be relatively different from the original code, and if equal weights are assumed for these two factors then a very good individual which is very different from the original program would be associated with a low fitness. To handle this issue this study proposes a new approach of using two fitness values. These two fitnesses should be calculated for each individual. The first one represents the number of passed test cases and the second one indicates the similarity of the individual to the original program. The number of passed test cases should be used as the primary fitness and in the situations where the passed fitness cases are equal, the selected individual will be chosen based on its similarity to the original program. Using this method, finding an individual which passes all the test cases would not be the end of GP evolution, but the individuals can continue to evolve to find candidates with less structural difference compared to the original program.

In [31] *Arcuri* discussed that in the context of Automatic Software Repair, using the number of passed test cases (fitness cases) for the unit under test might not give enough gradient for the GP evolution. He proposed to use the sum of all the output errors as the fitness value of each individual. However in our study this idea is not considered a contributing idea; and the reason to this is that the operation of software fix is quite different from other types of GP usages such as Symbol Regression which deal with numeric values; but in software repair the variable types, operators and behaviour of elements of methods have great diversity, and also the nature of software methods is usually very different from a simple numeric calculation. Therefore in the new proposed approach of this study the traditional number of passed test cases is used as the primary fitness measure.

Based on the proposed approach, a tool is developed to evolve codes written in C language. This tool is used to evolve a buggy code to a bug-free code and the results of this experiment are shown in section 4.3.

Chapter 4

Evaluation and Results

In this chapter the results of using the proposed methods in a number of sample projects will be presented. Each of the methods described in the previous chapter are implemented and applied to some sample problems and their results are represented in details in the following sections.

4.1 Test Optimization Case-Study

In this section the results of applying the proposed source-based test optimization approach (discussed in section 3.1) to a sample project will be presented. As for this level of the study only a high-level view of this experiment will be presented to illustrate the approach of applying the source-based technique to a sample project. A more detailed sample of source-based application will be presented in the next section as a part of compound risk model. A code analyser is developed based on our proposed technique. In the first step this tool is used to calculate *FR*, *FRD*, *CR* and *CRD* of all the components in the sample project. Figure 4.1 shows the *Component Risk Density diagram* of the sample project. In figure 4.1 it can be seen that *Comp1* has a relatively higher *Risk Density*

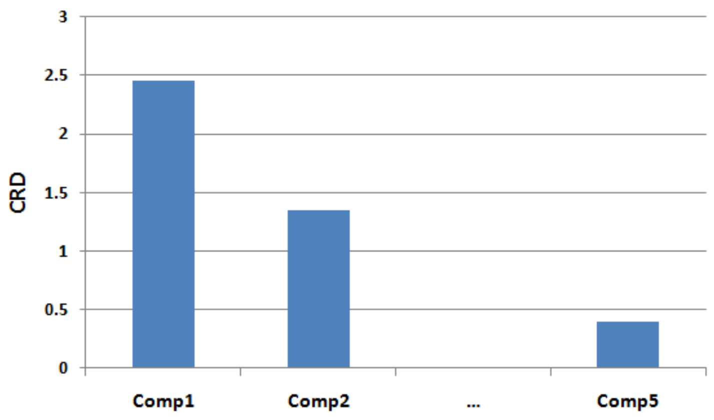

Figure 4.1: Components Risk Density

which implicates that it is more optimal to plan the test on this component. Figure 4.2 shows the *Function Risk Density* of the functions of *Comp1*.

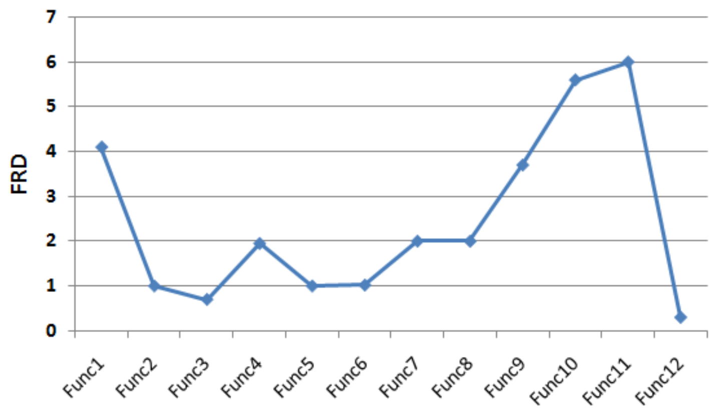

Figure 4.2: Functions Risk Density

Considering Figure 4.2 it can be seen that *Func11, Func10, Func1* and *Func9* have higher *FRD*s which indicates that they have a higher test priority.

Figure 4.3 shows the functions risk of *Comp1*. Comparing this figure with Figure 4.2 it can be seen that a function with lower *Function Risk* may have a higher *FRD*, and thus in a situation where the time is limited it is be a better choice for testing.

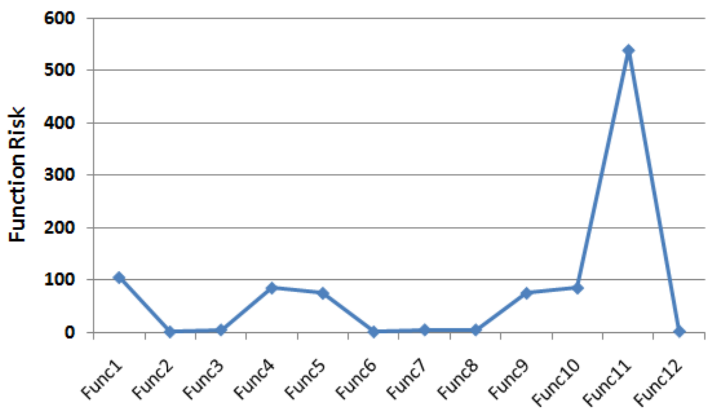

Figure 4.3: Functions Risk

4.2 Compound Risk Model Results

In this section the results of applying our compound risk model to a sample project will be presented. The sample project used in this study is a regression analysis system based on Genetic Programming. Figure 4.4 shows the components of this sample project and their dependencies[1].

The obtained values for $CR_1, CR_2, CR_3, IOFD$ and CFD are shown in Table 4.1. In this example all the weighting factors (except τ which is associated with the value 0.2) are given the same value of 1. The reason for choosing 0.2 for τ is that the dependencies between components are not total, meaning that only a fraction of the operations of the component under analysis are dependent on other components. In our study $\tau = 0.2$ is chosen which is based on the estimation of the average role of dependencies in the operation of the components.

[1]A more detailed view of the architecture of this project can be seen in Appendix A

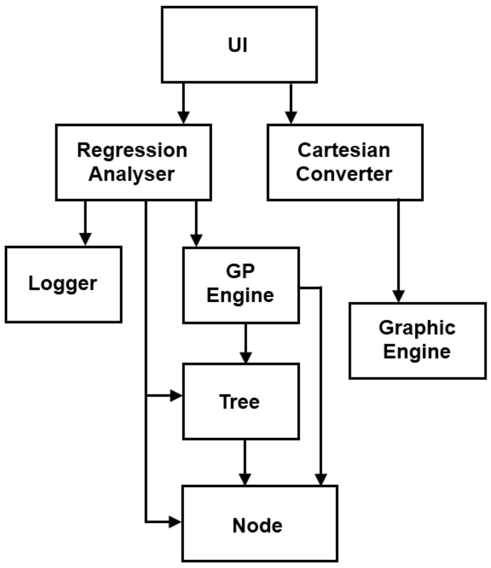

Figure 4.4: Dependency Diagram

Considering the values in Table 4.1 the Compound Risk Table for this example is shown in Table 4.2 and the compound risk diagram which is generated based on the compound risk table is shown in Figure 4.5.

By looking at Table 4.2 it can be seen that the component which is more probable to fail is the *GP Engine*. Considering that from the damage point of view *GP Engine* is

	CR_1 (Analysts Estimation)	CR_2 (Architectural Analysis)	CR_3 (Source-Code Analysis)	$IOFD$	CFD
Logger	20	2	41	0.1	0.32
UI	50	3	373	0.5	0.5
Cartesian Converter	60	10	231	0.4	0.5
Graphic Engine	40	6	45	0.5	0.6
Regression Analyser	60	8	362	1	1.1
GP Engine	80	10	1068	1	1.22
Tree	20	4	64	1	1.46
Node	10	4	28	1	1.76

Table 4.1: Metric Components

	CR	CFD	Damage Class
Logger	0.12	0.32	Negligible
UI	0.38	0.5	Marginal
Cartesian Converter	0.49	0.5	Marginal
Graphic Engine	0.27	0.6	Marginal
Regression Analyser	0.51	1.1	Critical
GP Engine	0.93	1.22	Critical
Tree	0.17	1.46	Catastrophic
Node	0.12	1.76	Catastrophic

Table 4.2: Compound Risk Table

classified as *Critical*, a good test plan should start by testing this component. Components *Tree* and *Node* are from the class *Catastrophic* but they have a very low relative probability of failure, thus in the test plan they are not considered as the highest priority. By looking at this table project manager can have a precise understanding of the risk specifications of different components and plan an optimal test and development.

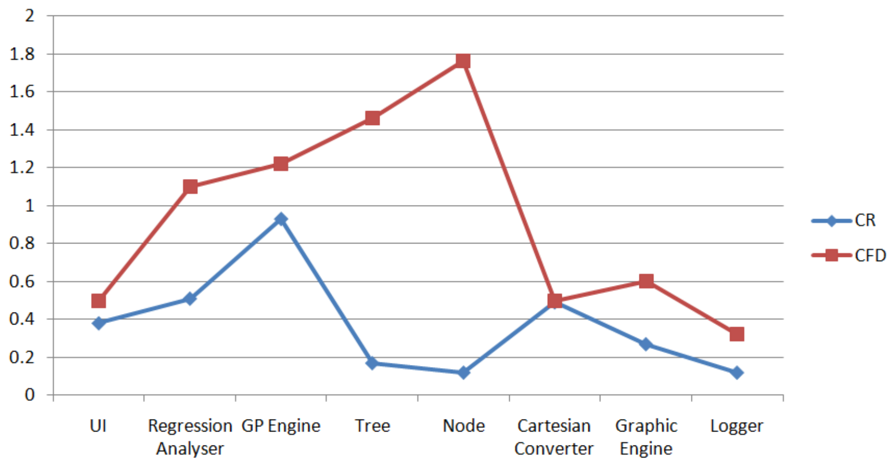

Figure 4.5: Compound Risk Diagram

To verify the results of proposed approach of this study, after the initial implementation the application is put under test and the number of failures which were caused by each component is measured. There are 60 different test-cases defined for this test with valid and invalid inputs. The result of the execution of each test-case and the failed component is shown in Appendix B. The record of each failure that caused the main operation of the application to stop is kept and its source is tracked. The result of this test provided the probability of each component to be the cause of a failure. This result can be seen in Table 4.3.

As it can be seen in Table 4.3 the highest failures were caused by the *GP Engine* which is also the most risky component (highest CR). The result of the test also matches with the obtained value for the second most risky component which is the *Regression Analyser.* This probability for *Cartesian Converter* is slightly different from the result of the risk analysis. The reason for that is because of the fact that this component is from a type which is more commonly developed and known by the developers of the application. In large projects the components of the subsystem under analysis would most likely be of the same type, thus the effect of this issue would be much less and almost negligible. Also by going back to table 4.1 it can be seen that using only one of the risk analysis methods can not provide enough precision. As an example by looking at CR_3 the component *Regression Analyser* has less chance of failure compared to *Cartesian Converter*, however

	CR	Cause of Failure Probability
Logger	0.12	0.017
UI	0.38	0.033
Cartesian Converter	0.49	0.017
Graphic Engine	0.27	0.033
Regression Analyser	0.51	0.083
GP Engine	0.93	0.100
Tree	0.17	0.000
Node	0.12	0.000

Table 4.3: Comparison of *CR* and the *Cause of Failure Probability*

the results of the tests in table 4.3 shows a different conclusion. This illustrates the idea that only one type of risk analysis can not be complete and it has to be combined with other risk analysis methods.

The proposed approach of this study can be used at any stage of the development of the software. In the early stages risk analysis can be done by excluding NCR_3 from the calculations; thus the risk metric will be calculated without considering code-dependent parameters. This risk metric can be used for basic planning and task association in the early stages of the development. After each subsystem development in each development cycle, code-dependent metrics (NCR_3) can be included in the Risk Analysis and as a result by identifying risky components, subsystem test and debug can be performed more precisely and more effectively. The high flexibility of our proposed approach (which is a result of using the weighting factors in calculating the metrics) makes it adjustable for different development teams with different experiences and skills.

Most of the proposed metrics in the proposed approach of this study are analytical metrics, meaning that there is a specific calculative approach to obtain them. This makes it possible to create an analysing tool which can be used to analyse a software and generate the metrics. This analyser tool can be designed to get the developed code and architectural diagrams as inputs and produce their corresponding risk factors (NCR_2 and NCR_3) and combine these values with the analysers' estimated metrics (NCR_1) to generate the Risk Model of the developing software. This will cause the risk analysis task to be very applicable and easy which results in a practical and precise approach to assist the creation of the development plan. Having this analyser tool as a part of the development environment will encourage the development team to provide the proper set of inputs and generate the risk model. This risk model will be completed upon each iteration of the development and can be used by test and development team to have the most optimized quality assurance process.

4.3 Automatic Bug-Fix Results

In this section the results of applying the proposed Genetic Programming Automatic Bug-Fix system to a sample problem will be presented. A GP evolutionary system is designed to evolve codes written in C language. In this system the process of evolution is executed as follows:

First a parser reads the original code and generates its parse tree. Then this parse tree is converted to a tree structure suitable for genetic programming operations. This individual is copied several times to create the population of the first generation. After applying the genetic programming operations and generating a new population each individual which is created in the form of genetic programming tree is converted back to a parse tree and then compiled and executed for each fitness case in a single thread. A time-out value is considered which is the limitation of the thread execution time and if the result of one execution does not come back in this time-out limit the thread is killed and the fitness evaluation will be considered as a failure for that fitness case. In case an individual is not compilable or it throws an exception such as division by zero, invalid memory reference, etc. the individual will be given the worst possible fitness and it will be eliminated upon next generation.

Using a C evolutionary system designed based on the proposed approach, an experiment is conducted with a sample code. In this experiment a buggy code is used which shown in Procedure 6. This code is written to return the quotient of dividing the input value by 5. The code shown in procedure 6 returns the wrong value over the cases where the result of dividing the input by 5 is an integer. A set of 25 fitness cases is used which

Procedure 6 Buggy code of finding the quotient of dividing a number by 5

```
int quotient() {
   int count;
   int i;
   int number;
   count = 0;
   scanf ("%d", &number);
   for (i = 1; i < number; i = i + 1) {
      if (i % 5 == 0) {
         count = count + 1;
      }
   }
   printf("Output: %d\n", count);
   return 0;
}
```

consists the numbers 1 to 25 and their proper output. Upon generation 83, by insering a mutated *for* loop, the evolutionary system manages to find an individual which could pass all the test cases. This individual is shown in procedure 7.

As it can be seen procedure 7 contains some redundant statements. Having an individual that can pass all the test cases, now the GP system uses the second level of fitness which is to make the candidate individual's structure as similar as possible to the original code. As the GP system continues to work, upon generation 97 it finds the ideal candidate which has a structure that is very close to the original code and passes all the test cases. This result is shown in procedure 8.

Figure 4.6 shows the change of fitness in the process of evolution. This figure only represents the fitness based on the number of passed fitness cases. It can be seen upon generation 83 an individual was found which passes all the test cases.

As it can be seen in procedure 8, the fix is a substitution of a $<$ with a $<=$. Since this change is modification inside the statements of the code other GP bug-fix technique mentioned in section 2.5 are not able to find this.

Using the approaches presented in section 2.5 the solution would show itself in form

Procedure 7 First fixed candidate of quotient method

```
int quotient() {
   int count;
   int i;
   int number;
   count = 0;
   scanf ("%d", &number);
   for (i = 1 ;i < number; i = i + 1) {
      for (i = 1 ;i <= number; i = i + 1) {  //mutated 'for' inserted
         if (i % 5 == 0) {
            count = count + 1;
         }
      }
   }
   printf("Output: %d\n", count);
   return 0;
}
```

Procedure 8 Final fixed code of quotient method

```
int quotient() {
   int count;
   int i;
   int number;
   count = 0;
   scanf ("%d", &number);
   for (i = 1; i <= number; i = i + 1) { //'<' is mutated to '<='
      if (i % 5 == 0) {
         count = count + 1;
      }
   }
   printf("Output: %d\n", count);
   return 0;
}
```

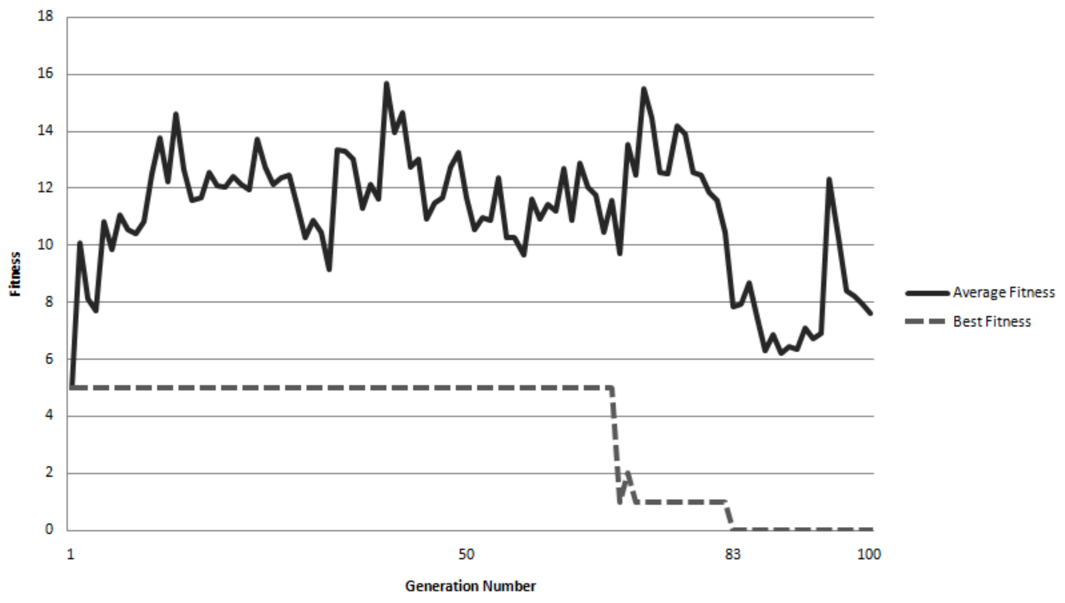

Figure 4.6: The fitness of the best individual and the average fitness of the population of Bug-Fix experiment

of adding a number of statements which can not achieve the simplicity of the fix applied by our proposed approach.

Chapter 5

Conclusion and Future Work

In this section a summary of the study in this thesis is presented and the plan for future work is discussed.

5.1 Conclusion

In this report we presented the details of our study on the subject of Automatic *Tools for Software Quality Assurance*. We started our study by analysing the risk factors of software components based on a source-based approach. After that by extending our research, we discussed that software project risks can be analysed from different points of view. We argued that each of these points of view identify the risk of a component by considering a subset of the factors that affect the quality and reliability of each component, and in order to have a more precise and reliable risk model we have to take all these factors into account which resulted in our compound risk model.

Using the compound risk model we could observe the effect of code and architectural structure of the application on its reliability. According to our source analysis approach having highly nested blocks of codes and high number of variable usages, function calls, etc. in different types of statements determine how complicated a source-code is. So it can be seen that by breaking big functions down to logical sets of functions decrease the level of riskiness of source-code. This result confirms the software design rules of modularizing the logic of application to increase the manageability and reliability of the code.

From the architectural point of view, according to our model having high number of dependencies between components results in propagation of risk damage. This result also confirms the current software design practices. Having a right abstraction, encapsulation and task association reduces the number of dependencies between objects and as a result reduces the chances of propagation of a failure in between the objects. Also having a wrong task association which causes an object to perform a highly complicated process increases the number of states of that object and creates complicated transitions between objects which increases the chance of failure of that component.

In small-scale projects the risk related specifications of each component can be estimated by an experienced project analyst, however in large projects the following factors make this estimation more complicated:

- numerous sub systems of the project
- different development teams working on different parts of the project
- out-sourcing
- multiple releases of the software product
- multiple platforms and development tools used in one project
- ...

Considering these factors that cause the project to be more complicated the project analysis team has to use some tools to aid them in their estimations about the different aspects of the components. We showed that our new technique can assist the project managers to plan the development by considering the risk factor of different components of the application which can be obtained by analysing the software project from different aspects.

In the genetic programming bug-fix experiment it could be seen that by defining specialized GP operators which are designed to reduce the amount of disruption in the individuals the optimal modification to fix bugs in a code can be achieved. It could also be seen that considering all the factors to keep the search space around the original code can assist the system to find the simplest and best solution.

5.2 Future Work

For future research the plan is to complete the code-analysis approach by including the logics of the application into the calculations. Intelligent techniques are also in the plan to be used in the code analyser. A training strategy can be applied which makes it possible to identify those kinds of risks associated with the development characteristics of each specific development team. Additionally a bug classification knowledge-base will be used to be attached to the genetic programming automatic bug-fix technique to make it possible to apply more intelligent fixes to more complicated set of defects. The long term goal is to design an approach to assist identifying the design problems of a given software and propose an alternative better design. To do that, software design patterns will be studied and their potentials will be examined to provide risk information about each component.

Appendix A

Detailed Class Diagram of The Regression Analyser Project

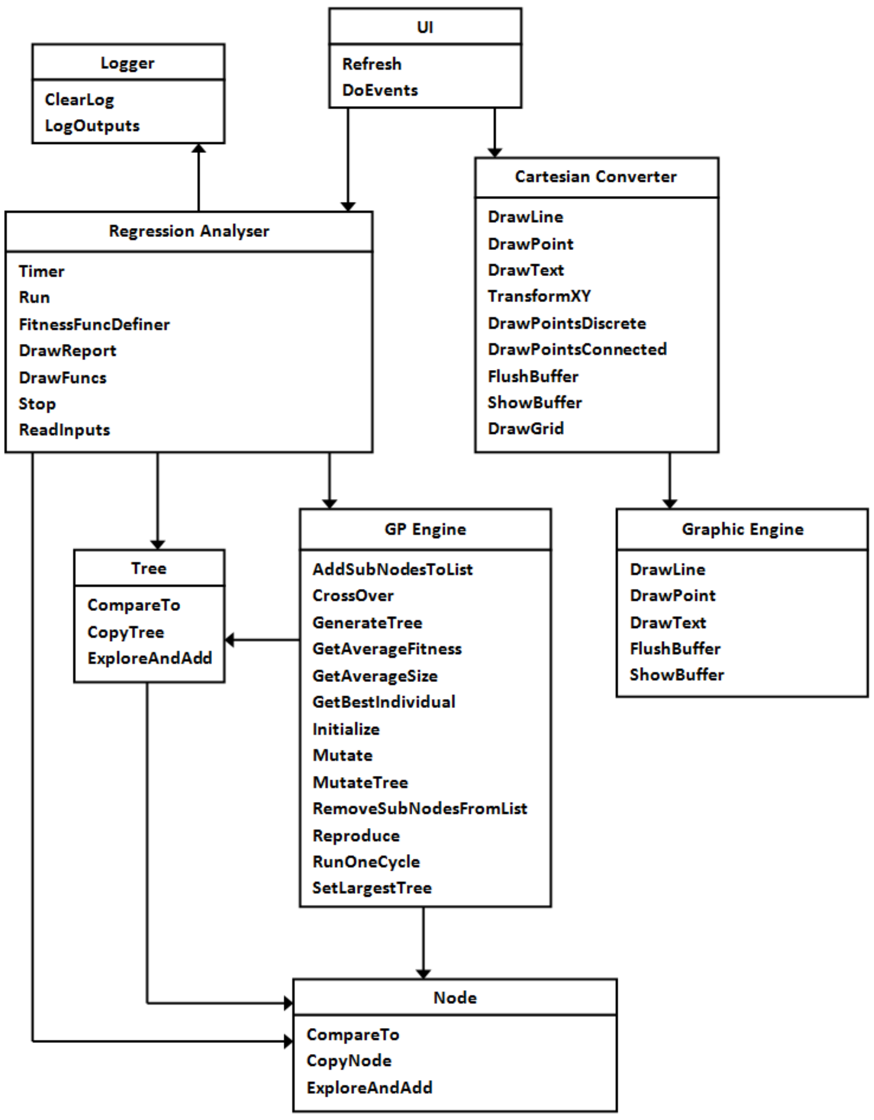

Figure A.1: Detailed Class Diagram of the Regression Analyser Project

Appendix B

Test results for Regression Analyser Project

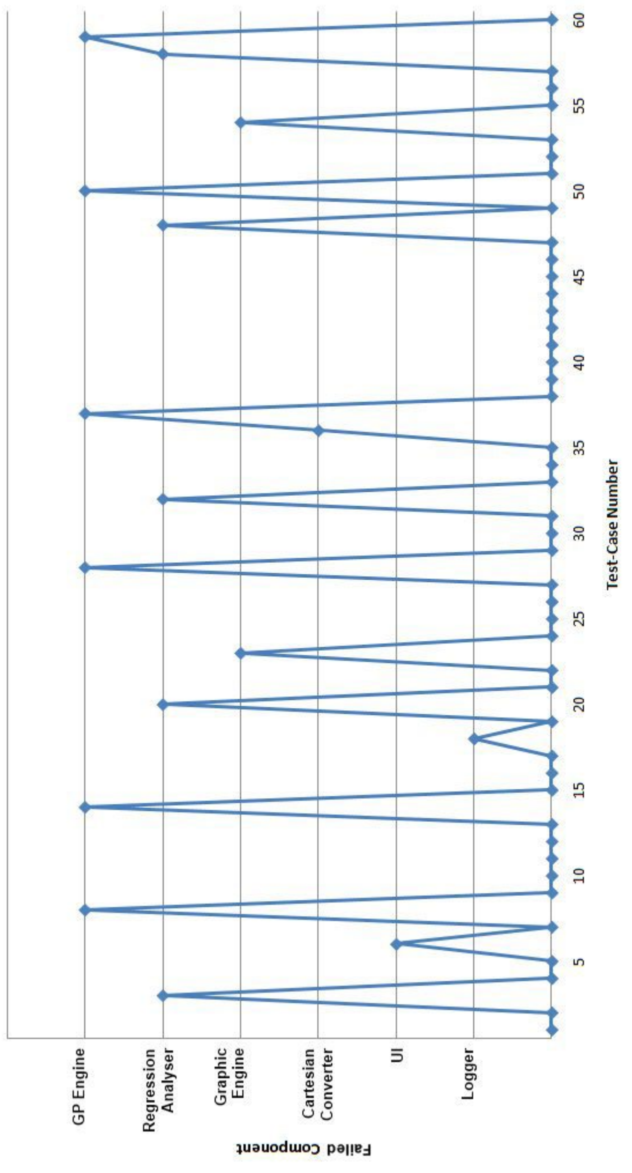

Figure B.1: The failed components in each execution of a test-case

References

[1] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *Proceedings of the 30th international conference on Software engineering*, pp. 111–120, 2008.

[2] P. R. Cox, "Elements of a software quality control program," in *Proceedings of the ACM '82 annual conference*, pp. 2–4, 1982.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing, final report.," tech. rep., National Institute of Standards and Technology, 2002.

[4] *NASA Safety Manual NPG 8719.13A*.

[5] Y. Tao, "A study of software development project risk management," in *Proceedings of the 2008 International Seminar on Future Information Technology and Management Engineering*, pp. 309–312, IEEE, 2008.

[6] B. W. Boehm, "Software risk management," in *Proceedings of 2nd European Software Engineering Conference, ESEC 89*, pp. 1–19, Springer, 1989.

[7] K.-H. Moller and D. Paulish, "An empirical investigation of software fault distribution," in *Proceedings of the First International Software Metrics Symposium*, pp. 82–90, 1993.

[8] M. Kaaniche and K. Kanoun, "Reliability of a commercial telecommunications system," in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 207–212, IEEE, 1996.

[9] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[10] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[11] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* McGraw-Hil, fifth ed., 2001.

[12] K. E. Emam and W. Melo, "The prediction of faulty classes using object-oriented design metrics, technical report nrc 43609," tech. rep., Nat'l Research Council Canada, Inst. For Information Technology, 2000.

[13] T. Khoshgoftaar, N. Seliya, and Y. Liu, "Genetic programming-based decision trees for software quality classification," in *Proceedings of 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 374–383, 2003.

[14] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Addison Wesly, third ed., 2004.

[15] V. Cortellessa, K. G. Popstojanova, K. Appukkutty, A. Guedem, A. Hassan, R. Elnaggar, W. Abdelmoez, and H. H. Ammar, "Model-based performance risk analysis," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 3–20, 2005.

[16] B. Beizer, *Software Testing Techniques.* International Thomson Computer Press, second ed., 1990.

[17] L. Bass, R. Nord, W. Wood, and D. Zubrow, "Risk themes discovered through architecture evaluations," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pp. 1–10, 2007.

[18] K. G. Popstojanova, "Architectural-level risk analysis using uml," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 946–960, 2003.

[19] S. Yacoub and H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 529–547, 2002.

[20] A. Deursen and T. Kuipers, "Source-based software risk assessment," in *Proceedings of the International Conference on Software Maintenance*, p. 385, 2003.

[21] A. Hosseingholizadeh, "A source-based risk analysis approach for software test optimization," in *Proceedings of the The 2nd International Conference on Computer Engineering and Technology (ICCET 2010)*, vol. 2, pp. 601–604, IEEE, 2010.

[22] W. E. Wong, Y. Qi, and K. Cooper, "Source code-based software risk assessing," in *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 1485–1490, 2005.

[23] A. Hosseingholizadeh and A. Abhari, "A new agent-based solution for wireless sensor networks management," in *Proceedings of the 2009 Spring Simulation Multiconference, 12th Communications and Networking Simulation Symposium (CNS)*, ACM, 2009.

[24] T. J. McCabe, “A complexity metrics,” *IEEE Transactions On Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[25] Y. Lei and J. H. Andrews, “Minimization of randomized unit test cases,” in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pp. 267–276, 2005.

[26] J. Carlos and B. Ribeiro, “Search-based test case generation for object-oriented java software using strongly-typed genetic programming,” in *Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO ’08)*, pp. 1819–1822, 2008.

[27] T. Nguyen, W. Weimer, C. L. Goues, and S. Forrest, “Using execution paths to evolve software patches,” in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 152–153, IEEE, 2009.

[28] S. Forrest, T. V. Nguyen, W. Weimer, and C. L. Goues, “A genetic programming approach to automated software repair,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954, 2009.

[29] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE 2009)*, pp. 364–374, IEEE, 2009.

[30] A. Arcuri and X. Yao, “A novel co-evolutionary approach to automatic software bug fixing,” in *IEEE World Congress on Evolutionary Computation*, IEEE, 2008.

[31] A. Arcuri, “On the automation of fixing software bugs,” in *Companion of the 30th International Conference on Software Engineering*, pp. 1003–1006, 2008.

[32] A. Hosseingholizadeh and A. Abhari, *Studies in Computational Intelligence Book Series: Software Engineering Research, Management and Applications 2010*, vol. 296, ch. eight: A New Compound Metric for Software Risk Assessment, pp. 115–131. Springer, 2010.

[33] L. Dobrica and E. Niemel, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.