A Bayesian Framework for Software Regression Testing

by

Siavash Mir arabbaygi (Mirarab)

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software maintenance reportedly accounts for much of the total cost associated with developing software. These costs occur because modifying software is a highly error-prone task. Changing software to correct faults or add new functionality can cause existing functionality to regress, introducing new faults. To avoid such defects, one can re-test software after modifications, a task commonly known as *regression testing*.

Regression testing typically involves the re-execution of test cases developed for previous versions. Re-running all existing test cases, however, is often costly and sometimes even infeasible due to time and resource constraints. Re-running test cases that do not exercise changed or change-impacted parts of the program carries extra cost and gives no benefit. The research community has thus sought ways to *optimize* regression testing by lowering the cost of test re-execution while preserving its effectiveness. To this end, researchers have proposed selecting a subset of test cases according to a variety of criteria (*test case selection*) and reordering test cases for execution to maximize a score function (*test case prioritization*).

This dissertation presents a novel framework for *optimizing* regression testing activities, based on a probabilistic view of regression testing. The proposed framework is built around predicting the probability that each test case finds faults in the regression testing phase, and optimizing the test suites accordingly. To predict such probabilities, we model regression testing using a Bayesian Network (BN), a powerful probabilistic tool for modeling uncertainty in systems. We build this model using information measured directly from the software system. Our proposed framework builds upon the existing research in this area in many ways. First, our framework incorporates different information extracted from software into one model, which helps reduce uncertainty by using more of the available information, and enables better modeling of the system. Moreover, our framework provides flexibility by enabling a choice of which sources of information to use. Research in software measurement has proven that dealing with different systems requires different techniques and hence requires such flexibility. Using the proposed framework, engineers can customize their regression testing techniques to fit the characteristics of their systems using measurements most appropriate to their environment.

We evaluate the performance of our proposed BN-based framework empirically. Although the framework can help both test case selection and prioritization, we propose using it primarily as a prioritization technique. We therefore compare our technique against other prioritization techniques from the literature. Our empirical evaluation examines a variety of objects and fault types. The results show that the proposed framework can outperform other techniques on some cases and performs comparably on the others.

In sum, this thesis introduces a novel Bayesian framework for optimizing regression testing and shows that the proposed framework can help testers improve the cost effectiveness of their regression testing tasks.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Ladan Tahvildari for her guidance throughout my research. Her support has made this research work possible.

Special thanks goes to Professor Kostas Kontogiannis and Professor Patrick Lam for their valuable time spent to read the thesis and for the valuable feedback they have provided.

I acknowledge my gratitude to all the researcher whom I have collaborated with, especially Professor Greg Rothermel, Professor Hyunsook Do, and Professor Amir Keyvan Khandani.

I would like to thank all the member of Software Technologies and Applied Research (STAR) Group for their moral support and valuable feedbacks.

Lastly, I would like to thank my mother for all she has done for me. I can not put my gratitude to her into words.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems are known to evolve with time and specially as a result of maintenance tasks [82]. Software maintenance is defined by IEEE Standard 1219 as [67]:

> The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

The presence of a long and costly maintenance phase in most software projects, specially those manipulating large systems, has persuaded engineers that software evolution is an inherent attribute of software development [82]. Moreover, maintenance activities are reported to account for high proportions of total software costs, with estimates varying from 50% [88] in the 80s to 90% in recent years [47]. Reducing such costs has motivated many advancements in software engineering in recent decades.

ISO/IEC 12207, a dominant standard in this area, identifies the objective of maintenance as "to modify the existing software product while preserving its integrity" [50]. The later part of the stated objective, preserving integrity, refers to an important issue raised as a result of software evolution. One need to ensure that the modifications made to the product for maintenance have not damaged the integrity of the product.

It is known both from theory and practice that changing a system in order to fix bugs or make improvements can affect its functionality in ways not intended. These potential side effects can cause the software system to regress from its previously tested behavior, introducing defects called *regression bugs* [51]. Although rigorous development practices can help isolate modifications, the inherent complexity of modern software systems prevents us from accurately predict the effects of a change. Practitioners have recognized such a phenomenon and hence are reluctant to change their programs in fear of introducing new defects. Researchers have tried to find ways of analyzing the impact of a change on different parts of a system [8] and predicting the effects. In absence of formal presentations of software systems, however, such attempts, although helpful, will not provide the required confidence levels.

Software maintenance remains a risky task unless we are able to find regression bugs, once they occur. Despite the introduction and adaptation of other verification methods (such as model checking and peer reviews), testing remains the main tool to find defects in software systems. Naturally, retesting the product after modifying it is the most common way of finding regression bugs. Such a task is very costly [84, 100] and requires great of organizational effort. This has motivated a great deal of research to understand and improve this crucial aspect of software development and quality assurance.

1.1 The Problem Statement

regression testing verifies that a software system has not regressed after modifications. It is the practice of retesting a program after changing it to assure that its previously tested behavior is preserved (non-regression). The goal of regression testing is to find regression bugs: defects introduced as a side effect of modifying programs. Although new test cases might be required to fully achieve such a goal, regression testing is mainly performed by re-running some subset of program's existing test cases. In this sense, Myers *et al.* describe regression testing as: " saving test cases and running them again after changes to other components of the program" [98].

Saving and re-running test cases from a previous version is reported to be a costly task [84, 100]. Testers face many challenges during the regression testing phase. To enable re-running, we must be able to either record or formally describe test cases. Moreover, as a program evolves, its corresponding test suite needs to be maintained to adapt to a potentially changed specification of the software system. *Test case maintenance* can reveal some test cases to be irrelevant with respect to the new version of the product, the so called *obsolete* test cases [84]. Even after removing obsolete test cases, many of the test cases exercise areas of code irrelevant to what has changed from the previous version. Re-running those test cases will only take extra effort with little chance of finding new faults. This motivates selecting a subset of test cases for re-execution in order to save some effort at no significant risk. After limiting the test cases to those related to changes, the number of selected test cases might still remain beyond what the organization affords to execute. In such situations, testers needs to choose test cases with higher priority and execute them.

A complete software regression testing process involves tasks pertaining to each of the described challenges. Considering a program that has changed from P_v to P_{v+1}, and having a test suite T_v created for and run on P_v, a regression testing process typically involves:

(i) Maintaining T_v to get T_v^m: identifying obsolete test cases and repairing or discarding them

(ii) Optimizing T_v^m to get T_v^o: selecting a subset of test cases and prioritizing the selected test cases

(iii) Running T_v^o on P_{v+1}, checking the results, and identifying failures

(iv) Creating a new test suite T_{v+1}^n, if necessary, to test P_{v+1}.

(v) Running T_{v+1}^n on P_{v+1}, checking the results and identifying potential failures

(vi) Aggregating T_v^o and T_{v+1}^n to get T_{v+1} and saving it for future testing.[1]

Each of the aforementioned tasks involves interesting research problems. This research work concerns merely the second step, as most other research efforts in this area do. How to find obsolete test cases, how to repair test cases, how to efficiently rerun test cases, how to create new effective test suites, are all interesting question that are out of the scope of this thesis. The problem of regression test suite optimization is a wide area which entails further fine-grained problems. The following first introduces a general form of the problem and then presents an instance of the general problem which is the focus of this research.

The regression test optimization problem can be defined in a general sense as:

Definition 1. *General Regression Test Optimization Problem:*
Given: a program P_v, its "maintained" test suite T_v^m, and a new version P_{v+1}
Problem: to find a test suite $T_v^o \sqsubseteq T_v^m$ such that it can most effectively, according to some criteria, find the regression bugs in P_{v+1}.

A test suite, in Definition 1 and hereinafter, is a tuple of test cases, the order of which indicates the execution sequence. If a test case t_1 appears before t_2 in the tuple, t_2 is not run unless t_1 runs. If A and B are two test suites, in our notation, $A \sqsubseteq B$ means $\forall\ x \in A \Rightarrow x \in B$. Notice that this notation does not imply anything as to the order of elements in A and B; hence, the order of test cases can be completely different between A and B.

The term *maintained* in Definition 1, as described before, means that obsolete test cases are dealt with before starting test optimization. More formally, it means that according to the specifications of P_{v+1}, which could be different from those of P_v, all the test cases in T_v^m are indeed expected to pass. Some obsolete test cases are those which pertain to features of the software that do not exist anymore. Such test cases should be removed as part of the test case maintenance. Others are those still relevant but incorrect due to changes in the specification of the software. Those test cases should be fixed to meet new requirements. Also, in Definition 1, a

[1] The superscript m stands for *maintained*, o for *optimized*, and n for *new*.

regression bug refers to the situation when a software functionality works as desired in P_v but is faulty in P_{v+1}.

The *criterion of effectiveness*, a main factor in Definition 1, specifies the goal of the test case optimization. The generality of this definition is due to the fact that no specific criterion is mentioned. Different products, testing processes, and organization policies can emphasize different goals for regression test optimization. In a safety critical project, for instance, any test case that has any chance of finding bugs must be executed. In the competitive markets of less sensitive products, on the other hand, organizations are usually willing to take the risk of discarding some test cases in favor of releasing the software sooner. This encourages an effectiveness criterion that intensively reduces the size of test suites. If an organization is uncertain about when to stop regression testing, effectiveness criteria that provide more flexibility can be helpful.

To address diverse expectations from regression test optimization, researchers have defined different instances of the problem. There are two predominant problem specifications in the literature: *test case selection* and *test case prioritization*. Test case selection problem, informally, is to optimize the regression test suite by eliminating a subset of test cases according to some criteria. Since many of the existing test cases can not, or according to some criteria, are not likely to, reveal faults even if executed, eliminating them can optimize the regression test suite. For example, those test cases unrelated to the changed or change-impacted parts of the code are not likely to reveal faults. By eliminating such test cases, one can reduce time required for the regression testing phase. Regression test selection does not concern the execution order of the resulting test suite. Test case prioritization problem, on the other hand, focuses on the order in which test cases appear in the final test suite (T_v^o).

The focus of this research work is on the test case prioritization problem. While terms and definitions regarding test case selection are introduced in Section 2.1.1, our problem statement focuses on test case prioritization.

1.1.1 Test Case Prioritization Problem

Speaking informally, the test case prioritization problem is to order test cases in a test suite such that a certain property is optimized for that ordering. The formal definition of this problem, widely accepted in the literature [111], could be rephrased as:

Definition 2. *Test Case Prioritization Problem:*
Given: a program P_v, its maintained test suite T_v^m, a new version P_{v+1}, and a function $f(T)$ from a test suite T to a real number,
Problem: to find an ordered test suite $T_v^o \in Perm(T_v^m)$ such that $\forall\, T \in Perm(T_v^m)$, $f(T) \leq f(T_v^o)$.

In Definition 2, $Perm(T)$ denotes the set of all permutations (orderings) of the test suite T. A Test suite A is a permutation of test suite B if and only if $A \sqsubseteq B$ and $B \sqsubseteq A$. The $f(T)$ function is a score function which reflects the goal of prioritization.

The $f(T)$ function assigns to a test suite a relative score that indicates how well it satisfies the goals of prioritization. Different goals can be pursued by testers when prioritizing test cases. They may wish to find bugs faster, cover more parts of code faster, find severe faults sooner, run easier-to-execute test cases earlier, and so on. The score function $f(T)$ assigns higher values to orderings which achieve an specific goal faster.

The prioritization goal that has been most extensively researched is that of early fault detection. Such a goal, informally, means a test suite is preferable if it finds bugs faster than others. Rothermel *et. al.* have formalized this goal using a measure called Average Percentage of Faults Detected (APFD) which could be thought of as the $f(T)$ function in the above definition [110, 111]. This metric averages the percentage of faults detected during the execution of a test suite. As such, it gives a number between 0 and 100 that indicates faster rate of fault detection if higher. The mathematical definition of this measure can be found in [111].

Test case prioritization for early fault detection has two main advantages. First, the cost associated with each bug is partially related to the time it is discovered. If a bug is found early in the regression testing process, debugging can start earlier and hence bugs could be fixed faster. In time-constrained situations (*e.g.,* with tight release schedules), this can have a huge impact because late detection of faults restricts the time available for bug-fixing and therefore some bugs can remain unfixed. Second, if the time constraints force testers to halt regression testing prematurely, early fault detection ensures that fewer faults are missed due to incomplete testing. This property is of special importance in situations where re-running all test cases is not feasible and hence a cut-off is likely to occur. Naturally, in this sense, test case prioritization and selection are similar to each other.

Early fault detection, as the goal of prioritization, can not be attacked directly as an optimization problem. Obviously, when optimizing a test suite for regression test execution, we are not aware of the existing faults and the test cases which reveal them. To achieve this goal, therefore, one needs to rely on the heuristics known to engineers regarding the behavior of faults and test cases. For example, one can assume that providing coverage for more parts of the code and as fast as possible is likely to improve the rate of fault detection, or that covering the fault-prone areas of the code faster can accelerate fault detection. Different heuristics, in addition to different algorithms of optimization, create different techniques of test case prioritization.

Finally, it is interesting to point out that test case prioritization and selection problems have much in common. If the goal of test case selection is merely to reduce the test suite size, that could be achieved by applying a prioritization technique and then cutting off the order-optimized test suite from a certain point. This solution is more flexible in that it provides a means of adjusting testing effort and available time. Test selection techniques typically do not control the number of selected test cases. Hence, if the resulting number of test cases is more than what testers can afford to execute, selection is not sufficient. On the other hand, test selection techniques provide a clear justification (according to a specified criteria of coverage, etc.) for eliminating test cases which prioritize-then-cut-off approach can not provide. It is also noteworthy that it is common to address these two families of problems in concert and/or feed the result of one to the other. For example, many researchers propose starting with test case selection (with criteria that prunes unrelated test case and retains relevant test cases) and then prioritize the resulting pruned test suite.

1.2 Thesis Contributions

Due to the probabilistic nature of fault detection process, proposed solution to the regression test optimization problem for enhanced fault detection all rely on heuristics. Many heuristics have been suggested in the literature and implemented using several optimization algorithms. However, the empirical studies indicate a gap between the optimal solution to the problem and any heuristic-based approaches. Each heuristic captures a different aspect of the fault detection phenomenon. One can conjecture that putting heuristics together would enable the consideration of more aspects and would hence result in better performance. This thesis introduces a novel framework that combines different heuristics into a single model aiming to increase fault detection power. The proposed framework not only uses many sources of information but also provides flexibility in which heuristics and measurements to use.

This dissertation makes the following contributions:

- It introduces a novel framework that uses Bayesian Networks (BN) to solve the regression test optimization problem by predicting the probability of test cases finding faults using multiple heuristics. This BN-based framework is designed for a generic software system and hence needs to be realized to get a specific implementation.
- It provides a code-based realization of the general BN-based framework. Software code is the only artifact this realization uses for heuristic measurements and hence the realization is applicable for most software systems.
- It reports on three sets of experiments that empirically evaluate the performance of the framework. One experiment concerns the effects of the parameters in the implementation of the framework on its performance; the two other compare its performance against existing techniques from the literature.

The novel BN-based framework, proposed by this dissertation, takes a probabilistic view of the problem. Such a framework is necessary, in the author's view, because the regression test optimization problem is inherently one in which a great deal of uncertainty exists. Modeling such systems using probabilistic tools sounds like a natural choice. The results from this line of research has proven this approach to be fruitful.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 provides a literature review of research related to this work. It first surveys the existing regression testing literature. Then, techniques of software measurement related to the topic of regression testing are examined. Finally the chapter gives an overview of the probabilistic modeling tools utilized in this research work is given.

Chapter 3 presents the general framework proposed by this thesis to the problem of regression test optimization. This chapter starts with an overview which explains the underlying ideas. Reading the overview should provide the readers with an overall understanding of the framework. Then, a process model is presented to detail how the general framework should be implemented to get a regression test optimization tool.

Chapter 4 introduces a code-based realization of the framework, where measurements come from the software code. The chapter first describes the motivations behind a code-based realization and why it is of particular importance. Using the path for realization provided by the previous chapter, it then describes in three different steps how the realization works.

Chapters 5 to 8 report on the results from a comprehensive empirical study on the performance of the code-based realization of the framework. Initially, Chapter 5 discusses the evaluation procedure of the experiments. The next following chapters present three different experiments, for each of which the design of the experiment, the threats to its validity, the obtained results, and a discussion on the results is given. The first experiment, Chapter 6, concerns the effects of parameters on the framework; the second one, Chapter 7, compares the framework against existing techniques and based on mutation faults; the third experiment, Chapter 8, conducts same comparison but using hand-seeded faults.

Finally, Chapter 9 finishes the thesis by drawing conclusions from the presented research work. It, first, gives a critical review of the presented framework, then future directions of research are pointed out, and finally a conclusion section finishes the thesis.

Chapter 2

Related Work

In this chapter, research areas related to the topic of this thesis are elaborated.

The subject to start with is that of the problem in question, “Software Regression Testing”. There exists an extensive body of research addressing this problem using many different approaches. This chapter takes a critical look at this line of research, trying to find strong points and ideas as well as the gaps. Through this examination, many terms and concepts related to software testing area will be introduced as well.

The second major area to discuss is “Software Measurement and Understanding”. One of the main goals of this research work is to provide a means of using different sources of information to assist regression testing tasks. The advances in software measurement and comprehension in recent decades have introduced new techniques of extracting useful information from many different types of software artifacts such as the requirement documents, the source/byte code, and the execution traces. This family of techniques, to which this document refers with the umbrella name of “Software Measurement and Understanding”, can assist regression testing tasks by providing useful information as to what needs to be tested more extensively. The ideas from this area of research have been used in many of the existing approaches to software regression testing problem.

Finally, as the title of this thesis suggests, a probabilistic modeling tool is used in this research work to attack the regression testing problem. While Chapter 3 elaborates on how and why probabilistic modeling and reasoning is used to address regression testing problem, here, an overall overview of probabilistic modeling concepts is presented. Specially, the powerful modeling technique of Bayesian Network (BN) is described in detail.

The rest of this chapter elaborates each of the three aforementioned areas.

2.1 Software Regression Testing

Research in regression testing spans a wide range of topics. Earlier work in this area investigated different environments that can assist regression testing (e.g., [15, 40, 129]). Such environments particularly emphasize automation of test case execution in the regression testing phase. For example, techniques such as capture-playback (e.g., [86]) have been proposed to help achieve such an automation. Furthermore, test suite management and maintenance have been addressed by much research (e.g., [58, 61, 116]). Measurement of regression testing process has also been researched extensively and many models and metrics have been proposed for it [38, 84]. Most of the research work in this area, however, has focused on test suite optimization.

Test suite optimization for regression testing, as described in Section 1.1, consists of altering the existing test suite from a previous version to meet the needs of regression testing. Such an optimization intends to satisfy an objective function, which is typically concerned with reducing the cost of retesting and increasing the chance of finding bugs (reliability). There exists a variety of techniques addressing this problem. Most of these techniques can be categorized into two families of *test case selection* and *test case prioritization.* Regression test selection techniques reduce testing costs by including a subset of test cases in the new test suite. These techniques are typically not concerned with the order in which test cases appear in the test suite. Prioritization techniques, on the other hand, include all test cases in the new test suite but change their order in order to optimize a score function, typically the rate of fault detection. These two approaches can be used together; one can start with selecting a subset of test cases and then prioritize those selected test cases for faster fault detection.

The rest of this section first looks into test case selection approaches from the literature and then touches upton existing techniques for test case prioritization.

2.1.1 Test Case Selection

Test case selection, as the main mechanism of selective regression testing, have been widely studied using a variety of approaches (e.g., [6, 9, 20, 49, 59, 60, 61, 108, 117, 120, 124]). In a survey of techniques proposed upto 1996, Rothermel and Harrold propose an approach for comparison of selection techniques and discuss twelve different family of techniques form the literature accordingly [109]. They evaluate each technique based on four criteria: inclusiveness (the extent to which it selects modification revealing tests), precision (the extent to which it omits tests that are not modification revealing), efficiency (its time and space required), and generality (its ability to function on different programs and situations). These four criteria, in principle, capture what we expect form a good test case selection approach. These four criteria inherently impose a trade-off situation where proposed techniques usually satisfy one of the criteria in expense of the others.

Main Approaches

An early trend in test selection research evolved around minimizing test cases selected for regression (e.g., [49, 61]). This approach, often called *test case minimization*, is based on a systems of linear equations to find test suites that cover modified segments of code. Linear equations are used to formulate the relationship between test cases and program segments (portions of code through which test execution can be tracked, e.g., basic-blocks or routines). This system of equations is formed based on matrices of test-segment coverage, segments-segment reachability, and (optionally) definition-use information about the segments. A 0-1 integer programming algorithm is used to solve the equations (an NP-hard problem) and find a minimum set of test cases that satisfies the coverage conditions. This approach is called minimization in the sense that it selects a minimum set of test cases to achieve the desired coverage criteria. In doing so, test cases that do cover modified parts of code can be omitted because other selected test cases cover same segments of the code.

A different set of approaches have focused on developing safe selection techniques. Safe techniques (e.g., [20, 59, 108, 117]) aim to select a subset of test cases which could guarantee, given certain preconditions, that the left-out test cases are irrelevant to the changes and hence will pass. Informally speaking, the aforementioned condition as described in [109] are: *i*) the expected result for test cases have not changed from the last version to the current version; and *ii*) test cases execute deterministically (*i.e.,* different executions results in identical execution path). Safe techniques first perform change analysis to find what parts of the code can be possibly affected by the modifications. Then, they select any test case that covers any of the modification-affected areas of the code. Safe techniques are inherently different from minimization techniques in that they select all test cases that have a chance of revealing faults. In comparison, safe techniques usually result in a larger number of selected test cases but also achieve a much better accuracy.

Many techniques are neither minimizing nor safe. These techniques typically use a certain coverage requirement on modified or modification affected parts of code to decide whether a test case should to be selected. For example, the so-called dataflow-coverage-based techniques (e.g., [60]) select test cases that exercise data interactions (such as definition-use pair) that have been affected by modifications. These selections techniques are different in two aspect: the coverage requirement they target and the mechanism the use to identify of modification-affected code. For example, Kung *et al.* propose a technique which accounts for the constructs of object-oriented languages [78]. In performing change analysis, their approach takes into account object-oriented notions such as inheritance. The relative performance of these selection techniques tend to vary from program to program, a phenomenon that could be understood only through empirical studies.

Cost Effectiveness

Many empirical studies have evaluated the performance of the test case selection algorithms (e.g., [20, 54]). In general, these empirical studies show that there is an efficiency-effectiveness (or inclusiveness-precision in [109]'s terminology) tradeoff between different approaches to selection (e.g., [54]). Some (such as safe) techniques reduce the size of test suite by a small factor but find most (or all) bugs detectable with existing test cases. Others (such as minimization techniques), reduce the size dramatically but can potentially leave out many test cases that can in fact reveal faults. Other techniques are somewhere in between; they may miss some faults but they reduce the test suite size significantly. The presence of such a tradeoff situation renders the direct comparison of techniques hard.

A meaningful comparison between regression testing techniques requires answering one fundamental question: is the regression effort resulting from the use of a technique justified by the gained benefit? To answer such a question one needs to quantify the notions of costs and benefits associated with each technique. To that goal, researchers have proposed models of cost-benefit analysis (e.g., [38, 85, 90]). These models try to capture the cost encountered as a result of missing faults, running test cases, running the technique itself including all the necessary analysis, etc. The most recent of all these models is that of Do et al [38]. Their approach computes costs directly and in dollars and hence is heavily dependent on good estimations of real costs from the field. An important feature of their model is that it can compare not only test case selection but also prioritization techniques. Most interestingly, it can compare selection techniques against prioritization techniques.

The existence of the mentioned trade-off has also encouraged the researchers to seek multi-objective solutions to the test selection problem. Yoo and Harman have proposed pareto efficient multi-objective test case selection [124]. They use genetic algorithms to find the set of pareto optimal solutions to two different representations of the problem: a 2-dimensional problem of minimizing execution time and maximizing code coverage and the 3-dimensional problem of minimizing time and maximizing both code coverage and history of fault coverage. The authors compare their solutions to those of greedy algorithms and observe that greedy algorithms surprisingly can outperform genetic algorithms in this domain.

Coverage information, a necessary input to most existing techniques, can be measured only if the underlying code is available and its instrumentation is cost-effectively possible. To be able to address more complex systems, where those conditions do not hold, some recent techniques have shifted their focus to artifacts other than code, such as software specification and component models (e.g., [9, 19, 99]). These techniques typically substitute code-based coverage information with information gathered from formal (or semi-formal) presentations of the software. Orso et al., for example, use component meta data to analyze the modifications across large component-based systems [99]. The trend in current test case selection research seems to be that of using new sources of information or formalizations of a software system to understand the impacts of modifications.

2.1.2 Test Case Prioritization

The regression Test Prioritization (RTP) problem seeks to re-order test cases such that an objective function is optimized. Different objective functions render different instances of the problem, a handful of which have been investigated by researchers. Besides targeted objective functions, the existing body of prioritization techniques typically differ in the type of information they exploit. The algorithm employed to optimize the targeted objective function, also, is another source of difference between the techniques.

Conventional Coverage-based Techniques

Test case prioritization is introduced in [122] by Wong *et al.* as a flexible method of selective regression testing. In their view, RTP is different from test case selection and minimization in that it provides a means of controlling the number of test cases to run. They propose a coverage-based prioritization technique and specify *cost per additional coverage* as the objective function of prioritization. Given the coverage information recorded from a previous execution of test cases, this coverage-based technique orders test cases in terms of the coverage they achieve according an specific criterion of coverage (such as the number of covered statements, branches, or blocks). Because the purpose of RTP in their work is selective regression testing, they compare its performance against minimization and selection techniques.

The coverage-based approach to prioritization is built upon by Rothermel *et al.* in [110, 111]. They refer to *early fault detection* as the objective of test case prioritization. They argue that RTP can speed up fault detection, an advantage besides the flexibility it provides for selective regression testing. Early detection makes faults less costly and hence is beneficial to the testing process. They introduce Averaged Percentage of Faults Detected (APFD) metric to measure how fast a particular test suite finds bugs. They also introduce many variations of coverage-based techniques, using different criteria for coverage such as branch coverage, statement coverage, and fault-exposing-potential.

These coverage-based techniques differ not only on the coverage information they use, but also on their optimization algorithm. When ordering test cases according to their coverage, a feedback mechanism could be used. Here, feedback means that each test case is placed in the prioritized order taking into account the effect of test cases already added to the order. A coverage-based technique with feedback prioritizes test cases in terms of the numbers of *additional* (not-yet-covered) entities they cover, as opposed to total number of entities. This is done using a greedy algorithm that iteratively selects the test case covering the most not-yet-covered entities until all entities are covered, then repeats this process until all test cases have been prioritized. For example, assume we have a system with six elements: $e_1 \dots e_6$ and the coverage relations between test cases and elements are as follows: $t_1 \rightarrow \{e_2, e_5\}, t_2 \rightarrow \{e_1, e_3\}, t_3 \rightarrow \{e_4, e_5, e_6\}$. According to a coverage-based technique, the first chosen test case is t_3 because it covers three elements, while the others cover two elements each. After selecting t_3, two test cases are left, both of which cover two elements. In the absence of feedback, we would choose randomly between the remaining two. However, we know that e_5 is already covered by t_3; therefore t_1 has merely one *additional* coverage, whereas t_2 has two. After adding t_3, we can update the model of coverage data such that the already tested elements do not effect subsequent selections. This allows choosing t_2 before t_1 based on its *additional* coverage. The notion of using *additional* coverage is what feedback mechanism provides; techniques employing feedback are often called *additional*.

Many empirical studies have been conducted to evaluate the performance of coverage-based approach [39, 43, 111], most of which use APFD measure for comparison. These studies show that coverage-based techniques can outperform control techniques (including random and original ordering) in terms of APFD but have a significant room for improvement comparing to optimal solutions. They also indicate that in many cases, feedback employing techniques tend to outperform their non-feedback counterparts, an observation which could not be generalized to all cases. Indeed, an important finding of all these studies is that the relative performance of different coverage-based techniques depends on the programs under test and the characteristics of its test suite. Inspired by this observation, Elbaum et al. have attempted to develop a decision support system (using decision trees) to predict which technique works better for what product/process characteristics [44].

Many research works have enhanced the idea of coverage-based techniques by utilizing new sources of information. Srivastava *et al.* propose the Echelon framework for change-based prioritization. Echelon first computes the basic blocks modified from the previous version (using binary codes) and then prioritizes test cases based on the number of additional modified basic blocks they cover [115]. A similar coverage criteria used in the context of aviation industry called Modified Condition/Decision Coverage (MCDC) is utilized in [71]. Elbaum *et al.* use metrics of fault-proneness, called fault-index, in order to guide their coverage-based approach to focus on the parts of code more prone to containing faults [43]. Recently, in [69], Jeffery and Gupta propose incorporating to prioritization a concept extensively used in test selection called *relevant slices* [77], modified sections of the code which also impact the outcome of a test case. Their approach prioritizes test cases according to the number of relevant slices they cover. Most recently, Zhang *et al.* propose a technique which could incorporate varying test coverage requirements [127] and prioritize accordingly. They work also takes into account different costs associated with test cases.

Recent Approaches

Walcott [118] *et al.* formulate a time-aware version of the prioritization problem in which a limited time is available for regression testing and also the execution time of test cases are known. Their optimization problem is to find a sequence of test cases that could be executed in the time limit and also maximize speed of code coverage. They use genetic algorithms to find solutions to this optimization problem. Their objective function of optimization is based on summations of coverage achieved, weighted by execution times. Their approach could be thought of as a multi-objective optimization problem where most coverage in minimum time is required.

In [87], authors investigate the use of metaheuristic and evolutionary algorithms (such as hill climbing and genetic algorithm) for test case prioritization. Their approach focuses on maximizing rate of achieved coverage using the search algorithms. Their objective is not to increase the rate of fault detection but rate of achieved code coverage. As such they do not investigate the rate of fault detection in their work.

All the code coverage-based techniques assume the availability of source/byte code. They also assume that the available code can be instrumented (see Section 2.2.3 for more information) to gather coverage information. These conditions do not always hold. The code could be unavailable or excessively expensive to instrument. Hence, researchers have explored using other source of information for test case prioritization.

Srikanth *et al.* have proposed PORT framework [63] which uses four different requirement-related factors for prioritization: customer-assigned priority, volatility, implementation complexity, and fault-proneness. Although the use of these factors is conceptually justifiable and based on solid assumptions, their subjective nature (especially the first and third factors) make the outcome dependent on the perceptions of customers and developers. While it is hard to evaluate or rely on such approaches, it should be understood that it is the subjective nature of requirement engineering that imposes such properties. Also, their framework is not concerned with specifics of regression testing but prioritization in general.

Bryce et al. have proposed a prioritization technique for Event-Driven Software (EDS) systems. In their approach, the criteria of *t-way interaction coverage* is used to order test cases [16]. The concept of interactions is defined in terms of events and the approach is tested on GUI-based systems and against traditional coverage based systems. Based on a similar approach, Sampath et al. target prioritization of test cases developed for web applications. Their technique prioritizes test cases based on different criteria such as test lengths, frequency of appearance of request sequences, and systematic coverage of parameter-values and their interactions [113].

Taking a different approach from coverage-based techniques, Kim and Porter propose using history information to assign a probability of finding bugs to each test case and prioritize accordingly [73]. Their approach, inspired by statistical quality control techniques, can be adjusted to account for different history-based criteria such as history of execution, history of fault detection, and history of covered entities. These criteria, respectively, give precedence to test cases that have not been recently executed, have recently found bugs, and have not been recently covered. From a process point of view, history-based approach makes the most sense when regression testing is performed frequently, as opposed to a one-time activity. Kim and Porter evaluate their approach in such a process model (*i.e.,* considering a sequence of regression testing sessions) and maintain that comparing to selection techniques and in the presence of time/resource constraints, it finds bugs faster.

Most recently, Qu *et al.* use the history of test execution for black-box testing and build a relation matrix between test cases [105]. This matrix is used to move the test cases up or down in the final order. Their approach also includes some algorithms for building and updating such a matrix based on outcome of test cases and types of revealed faults.

In addition to research works addressing the stated prioritization problem directly, there are research closely related to this area but from different perspectives. Saff and Ernst use behavior modeling to infer developers' beliefs and propose a test reordering schema based on their models [112]. They propose running test cases continuously in background while software is being modified. They claim their approach leads to reducing the wasted time of development by approximately 90Leon and Podgurski compare coverage-based techniques of regression testing with another family called distribution-based [83]. Distribution-based approaches [33] look at the execution profile of test cases and use clustering techniques to locate test cases that can reveal faults better. The experiments indicate that distribution-based approach can be as efficient or more efficient compared to coverage-based. Leon and Podgurski, then, suggest combining these two approaches and report achieved improvement using that strategy.

2.1.3 Critical Overview

Taking a critical look at the literature of regression test optimization problem helps identify interesting research challenges.

The results from the aforementioned empirical studies indicate that no single approach is able to provide a complete solution to regression test optimization problem. The discussion regarding selection techniques in Section 2.1.1 has revealed that there exist important trade-offs between the approaches in terms of cost-effectiveness. Such trade-offs should be ideally made by the engineers that work on a particular project and based on the characteristics of their environment/product. A desired property for any regression test optimization is to provide flexibility on the required testing effort; test selection techniques lack in that regard. Test case prioritization techniques take a step toward that flexibility by providing a means of adjusting the costs and benefits of test execution. These approaches, which are inherently heuristic-based, order test cases based on their *estimated importance*. Testers can use such a prioritized order to spend their effort on running the most important test cases.

To estimate the importance of test cases, the prioritization approaches rely on information that could be gathered from the software system. As such, any prioritization technique faces two major challenges. One is the availability of the required data. Since not all measurement techniques are doable in all environments, a technique that relies strictly on one particular source of information can easily become unapplicable. That is the reason why recent research in this area has focused on developing techniques that rely on different sources of information. Even if a measurement is available, it is not necessarily effective for all software systems. Empirical studies on existing approaches have all shown that various techniques, differing mainly in the utilized sources of information, are suitable for various programs. That has encouraged researcher to investigate the use of different information for prioritization (e.g., [16, 112, 63]). Some have even tried automatic identification of a technique that works better on a specific program [44]. More importantly people have tried mingling different heuristics together and reported improved performances [43, 83, 87]. Nonetheless, the proposed techniques are far from being stable across all investigated programs. There is also a gap between the theoretical optimal solutions to the problem and those of the proposed techniques. These two factors indicate that more regression test optimization approaches are indeed needed. Such approaches would ideally be able to use multiple sources of information for their prediction and also provide flexibility in used measurement techniques.

2.2 Software Measurement and Understanding

Software measurement and understanding are two areas of software engineering which have seen a great deal of emphasis and advancement recently. Many ideas from both of these research areas are relevant to the subject of this research work. This chapter provides necessary information about aspects of these areas which are more related to this research work.

Fenton, in his influential book on software metrics [48], argues that the field of software engineering lacks and can benefit from more rigorous engineering approaches, for which measurements and metrics are fundamental necessities. There, he gives the following definition for measurement :

> *Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real word in such a way as to describe them according to clearly defined rules.

In the same book, Fenton mentions that one of the main ways in which engineers can benefit from measurement is to “measure attributes of existing product and current processes to make predictions about future ones”. Such prediction models can be of great help to address the problem of regression testing. One can use prediction models to estimate attributes that are believed to relate to the outcome of regression testing and use the estimations to plan regression testing (*i.e.*, select or prioritize test cases).

The choice of measurement techniques to use and metrics to compute is an important issue. It mainly depends on what information are related to the regression problem and also what is (cost efficiently) available. Based on the definition of the regression testing problem presented in Section 1.1, some related measurement techniques can be identified. Modification of software artifacts is the main event triggering introduction of regression faults. Therefore, it is only natural to look at the evolution history of the product and identify the modified parts which should be the focus of regression testing. Software quality measurement is another research area conceptually related to regression testing. Quality metrics can be utilized to spot fault-prone areas of code. One can re-test those areas more extensively in the regression testing phase. Most importantly, measuring test coverage information provides crucial knowledge for regression testing. Such information help to plan regression testing such that a desirable coverage of some artifacts is achieved. Using techniques of runtime analysis such as byte code instrumentation, one can measure test case code coverage. Code coverage information constitutes the basis of most existing regression testing techniques.

The rest of this section will touch upon each of the above areas.

2.2.1 Change Analysis

The notion of change in software, sometimes called software evolution, has been widely studied and much has been written about it (e.g., [27, 82]). The area of software regression testing heavily depends on change analysis in software. Two main questions asked in this context are: *i*) what parts of software have changed, and *ii*) what parts are impacted by those changes. Software evolution is a wide area of research (for the ongoing research see [92]); here, the focus is merely on aspects of it that help answering the two aforementioned questions.

Regarding the first question, researchers have developed software differencing techniques and tools that detect changes between two versions of a program. Various differencing techniques use different representations of software. Typically, they use some heuristics to match as many elements (in their specific representation) as possible and mark the rest of elements as changed.

On the simplest form, one can look at source code as a text file and use lexical differencing tools, such as GNU diff tool [34], to compare two text files line-by-line. The wide availability of lexical differencing tools has rendered this approach as one of the prominent ways of code differencing in practice. The drawback is that this method ignores the structure of the code (from a software perspective) and hence the results are not accurate by no measure. More sophisticated representations of software, reflecting its structure, have been hence targeted by other approaches. Researchers have proposed differencing algorithms based on software representations such as the Abstract Syntax Tree (AST) (e.g., [123]), Control Flow Graph (CFG) (e.g., [2, 66]), and program dependency graph (e.g., [68]).

The binary code have also been proposed as a suitable representation of software for differencing purposes. BMAT (Binary Matching Tool), for example, performs matching on data blocks and uses some heuristics to find most matched blocks [119].

Sandmark [114], a watermarking tool, has also implemented several byte-code differencing algorithms that can compare methods or entire classes [26]. Sandmark implements many diff algorithms based on a variety of criteria such as longest common subsequence (LCS) of instructions, positional relationship between variable uses, number of matching basic blocks, maximum common subtree of two method CFG's, and LCS between the Constant Pools of classes.

Regarding the second question, that of finding change impacts, a huge body of research exist that directly or indirectly addresses this question. Change impact analysis, according to [7], is the task of *identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change.* There, authors classify change impact analysis techniques into two categories of source-based (dependency) analysis and traceability-based analysis. The existing techniques of regression testing are mostly code-based and as such use the first category of change impact analysis techniques. Here, a brief description of change impact analysis techniques most related to regression testing is presented; a comprehensive perspective on the research in this area is presented in [8].

Research on regression testing techniques has involved change impact analysis from early years. Harrold and Soffa have introduced a technique for analyzing the change effects within a module [60]. They use an intermediate representation of the code to detect structural and nonstructural changes. Their approach uses data-flow graphs to identify the affected definition-use pairs and/or sub paths. In addition to data-flow graph, control flow graph has been used by many approaches to identify the affected control paths inside a module (e.g., [80, 104]). Abstracting this notion to module interactions, White and Leung propose traversing the call graphs in order to find a set of modules possibly affected by a change, the so-called firewall of that change. Their work also encompasses a data-based definition of firewall concept which uses the data flow graph to find affected modules due to coupling through global data. The notion of firewall is also extended to the object-oriented languages in [78] to account for OO language structures such as inheritance and polymorphism.

In a different trend, researchers have explored using the history of co-changes to draw conclusions regarding change propagation (e.g., [130]). The idea behind these techniques is that if two entities change together frequently in the past, they are likely to change together in the future as well. Mirarab *et al.* have extended these approaches to account for both dependency analysis and history of change [95].

Finally, it should be noted that the research in this area is not by no means limited to the aforementioned topics. Change analysis is an ongoing area of research that gains more importance as software systems become more and more complex and interconnected.

2.2.2 Quality Measurement

Measuring the quality of software artifacts has been a major push in the software engineering area. Software quality can be defined from different view points, some of which are presented in [72]. One prominent view is that of Crosby [30], where quality is viewed as *conformance to requirements*. This definition inherently puts the emphasis on the notion of *defect* (non-conformance with requirements) as the primary indicator of the lack of quality. Quality measurement, in this paradigm, needs to focus on defects-related attributes. Measuring the presence of actual defects in the current software system is a natural way of seeking that objective. Many of the most-practiced quality assurance activities such as testing, code review, and dynamic analysis, take this approach and essentially try to find and/or measure bugs. A different but equally important aspect of defect-related measurements is defect prediction. Engineers are interested to know how much, where, and why defects can potentially occur in the system. Such measurements provide a foundation of information upon which decisions about software engineering tasks can be both planned and performed better [102]. In this paradigm, many different attributes of the software system are of interest because they can potentially be related to defects. For example, if a software system lacks "maintainability", during its evolution, the introduction of defects are likely.

The quality attributes of software systems are often divided to *internal* and *external*. *Internal* attributes are those which can be measured purely in terms of the product, process, or resource itself. *External* attributes are those which can only be measured with respect to the relations with environment [48]. External quality attributes, such as reliability and maintainability, are what usually interests costumers and managers. However, they are inherently hard to define and measure. Internal attributes, on the other hand, are easier to define precisely and measure accurately, but hard to interpret. This dilemma has encouraged the researchers to investigate the correlation between these two sets of attributes, in hope of building prediction/estimations models that use internal attributes to predict or estimate external quality attributes.

Reliability is one of the external quality attributes which is of particular interest to many stakeholder of any product. From testers' perspective, it is helpful to know which parts of the code are less reliable (or more fault-prone) so that the testing effort can be focused on those areas. In the regression testing, specifically, knowing fault-prone areas of code can be of great help because the test cases that cover those areas can be given higher priority. Fault-proneness predictions have been attempted using models that correlate the internal attributes (*i.e.*, structural design properties) of the available software system to fault-proneness. The following section describes such models.

Fault-proneness Models

Fault-proneness of a module (e.g., a class in an object-oriented system) is the number of defects detected in that module. Fault-proneness models are systematic ways of predicting this attribute. Conceptually, they estimate the probability that a software module has defects. In fact, measurement of fault-proneness is an indirect way of looking at reliability. Many techniques has been proposed for building such models , most of which use regression analysis [1] or AI-inspired techniques to exploit the software structural measures as potential indicators of fault-proneness.

Structural measures of code or design are ways of analyzing the structure of the software artifacts in terms of the interaction between elements (*i.e.*, modules, classes, methods) or their hierarchy. There have been a great deal of effort to introduce structural measures that correspond to external qualities of the system, mainly reliability and maintainability. Starting with traditional functional programming, many metrics has been introduced to measure different aspects such as: lexical tokens [56], control graph [91], and inter-connection among statements or functions [65]. With the advent of object-oriented paradigm, new metrics were introduced to capture the special characteristic of OO languages [14, 21, 89, 121]. These metrics measure properties related to concepts such as complexity [64], coupling [12], cohesion [11], inheritance [23], and polymorphism [4]. Also, many empirical studies have been conducted to examine if the proposed metrics (or metric suites) are related to the faultiness of the code [3, 18].

Many of these empirical study have focused on Chidamber and Kemerer (CK) suite of metrics [21]. This suite consists of six different metrics: WMC (Weighted Methods per Class), the number of all methods (member functions) defined in each class; DIT (Depth of Inheritance Tree), the number of ancestors of a class; RFC (Response For a Class), the number of functions directly invoked by methods of a class; NOC (Number Of Children), the number of direct descendants for each class; CBO (Coupling Between Object classes), the number of classes to which a given class is coupled (*i.e.*, it uses their methods and/or fields); LCOM (Lack of Cohesion on Methods), the number of pairs of methods without shared fields, minus the number of pairs of methods with shared fields. Basili *et al.* modified these definitions slightly to fit C++ language and then formalized six hypotheses (one for each metric) that captures the potential relation between the metrics and the fault-proneness of the code [3]. The results they obtained validated some of those hypotheses, did not prove some, and contradicted with the hypotheses in one case (NOC).

Based on the aforementioned studies, many other empirical studies have been conducted to evaluate the power of fault-proneness prediction models, using different metrics specially those of CK suite [13, 32, 45, 55, 97]. A comprehensive account of those studies is given in [10]. Most of the empirically evaluated models use regression analysis techniques (such as logistic or linear regression) for prediction. These

[1] a widely used mathematical method to predict an unknown variable based on one or more known variables (no relation to regression testing).

techniques are, in principal, statistical methods of relating Independent Variables (IV), quality metrics in our case, to Dependent Variables (DV), i.e., fault-proneness. There are two types of regression analysis: univariate - those which evaluate the relation of merely one IV to the DV-, and multivariate - those which combine multiple IVs to predict a DV. To use multivariate models, one should "train" the model on a second program (hopefully built by the same development team and process) and apply the potentially biased model to the system in question. This costly step is needed because building a multivariate regression model is essentially a learning task which calls for a learning data set. Briand *et al.* have empirically evaluated the application of multivariate models, when used across software projects [13]. Their experiments show that using models from one system for another significantly drops its performance. However, they argue that they are still good prediction model of fault-proneness, the output of which should be treated with caution.

Another option for fault-prediction is to build models merely based on one metric. Although most of the works in the literature perform univariate analysis, the author is not aware of studies which build univariate prediction models and evaluate their performance. A univariate model can be a pre-determined function of one metric and hence does not need to be trained on a second data set (program). The univariate analysis from literature (e.g., from [13] and [55]) indicate that measures of complexity and coupling are better indicators of fault-proneness. Moreover, an empirical study on the relation between APFD measure and software metrics presented in [42] suggests same metrics can be of great help for early fault-detection.

To wrap up this discussion, it is important to mention how one can use the results of predictive models. Once a prediction model is built, and after validating that it is a fairly accurate model, the next step is to make decisions based on the output of that model. The main promise of predicting fault-proneness is that the results can help quality assurance activities (e.g., testing) to focus on fault-prone parts of the code. It will be helpful for testers to have techniques that use the results of fault-proneness predictions to assist their tasks.

2.2.3 Test Coverage Measurement

The notion of *test coverage* is an indispensable part of testing practices and more so of the academic research on the subject. A major part of research in testing area concerns ways of generating test suites that achieve specific criteria of coverage (*e.g.,* [53, 103]). From test generation perspective, one needs to build a test suite that is adequate according to a coverage criterion [128]. The notion of coverage, however, is not restricted to test case generation. Regression testing techniques typically engage coverage information extensively as well. In this context, we are interested to measure the coverage of test cases that already exist. Although coverage can be defined regarding various artifacts involved in software life cycle (most notably requirements), it is most commonly defined in terms of code constructs.

Code coverage measurement is the task of identifying code constructs that are exercised through the execution of a test case. The promise behind using any measure of code coverage is that the more code coverage a test suite achieves, the more confidence it provides regarding the reliability of the system under test. If we assume a deterministic model of software behavior, where an specific input always results in the same execution path, the coverage of a test case remains unchanged from a run to another. Such data provide a dependable characteristic of the test case and can be used in different testing activities to understand test cases better.

Targeting specific code constructs yields different coverage metrics. Researchers have defined many criteria of coverage such as branch coverage, basic block coverage, and definition-use coverage. Statement coverage, for example, indicates whether a particular statement of the code (or an encoding of it such as byte code) is executed by a test case.

To measure coverage of a test case, one can trace the execution path of the program while running the test case, a task known as program profiling [79]. That could be done through modifying the code by inserting some additional code that records events occurring during the execution. The additional code is called *instrumentation code* and the process is referred to as *code instrumentation.* Code instrumentation is an automatic task that could be performed at different stages of programs' compilation. Instrumenting executable files directly has the merit of not affecting system tools such as compilers or linkers. It also facilitates a detailed measurements and simplifies the whole process because programs need not recompilation. Instrumenting executable files, in general, has three steps. First, the original program needs to be analyzed in order to capture the structure of the code. The result of this analysis is then used to create the instrumentation code and find the places where it should be inserted. Finally, the byte code is modified by inserting the instrumentation code such that the program's behavior is preserved. Once the byte code is instrumented, test cases could be executed on it; the recorded traces of the events can be then used to infer the coverage of each test case.

In the context of regression testing, coverage information from the previous version of software is usually used. To measure coverage of a test case one needs to instrument and then execute it. That defeats the purpose of regression testing optimization which is to avoid re-running unnecessary test cases. To deal with this problem, researchers have conjectured [107] and evaluated [41] that the coverage data gathered from the previous version, if available, can be a reasonable estimate of current coverage data. Therefore, coverage-based regression techniques assume when running test cases on a previous version, the byte code had been instrumented and the coverage information had been gathered. The available coverage data can guide the regression test optimization of the current version.

2.3 Probabilistic Modeling and Reasoning

The probability theory provides a powerful way of modeling systems. It is specially useful for situations where the effect of events in a system are not fully predictable and a level of uncertainty is involved. The behavior of large complex software systems are sometimes hard to precisely model and hence probabilistic approaches to software measurement have gained attention.

In the center of modeling a system with probability theory is to identify events that can happen in the system and model them as random variables. Moreover, the distribution of these random variables also need to be estimated. The events in the real systems and hence the corresponding random variables can be dependent on each other. Bayes theorem provides a basis for modeling the dependency between the variables through the concept of conditional probability. The probability distribution of random variables could be conditioned on others. This makes modeling systems more elaborate but also more complex. Different modeling techniques have been developed to facilitate such a complex task.

Probabilistic graphical model are one family of such modeling techniques. A probabilistic graphical models aims to make modeling system events more comprehensible by representing independencies among random variables. A probabilistic graphical model is a graph in which each node is a random variable, and the missing edges between the nodes represent conditional independencies. Different families of graphical models have different graph structures. One well-known family of graphical networks, used in this research work, is Bayesian Networks.

2.3.1 Bayesian Networks

Bayesian Networks (BN) (a.k.a "Belief Networks" and "Bayesian Belief Networks") are a special type of probabilistic graphical models [101]. In a BN, like all graphical models, nodes represent random variables and arcs represent probabilistic dependency among those variables. The missing edges from the graph, hence, indicate that two variables are *conditionally independent*. Intuitively, two events (*e.g.,* variables) are conditionally independent if knowing the value of some other variables makes the outcomes of those events independent. The conditional independence is a fundamental notion here because the idea behind the graphical models is to capture these independencies.

What differentiates a BN from other types of graphical models (such as Markov Nets [74]) is that it is a Directed Acyclic Graph (DAG). That is, each edge has a direction and there should be no cycles in the graph. In a BN, in addition to the the graph structure, the Conditional Probability Distribution (CPD) of each variable given its parent nodes should be specified. These probability distributions are often called the "parameters" of the model. The most common way of representing CPDs is using a table called Conditional Probability Distribution Table (CPT) for each variable (node). Each possible outcome of the variable forms a row, where each cell gives the conditional probability of observing that outcome, given a combination of the outcomes of the parents of the node. That is, these tables include the probabilities of outcomes of a variable given the values of its parents.

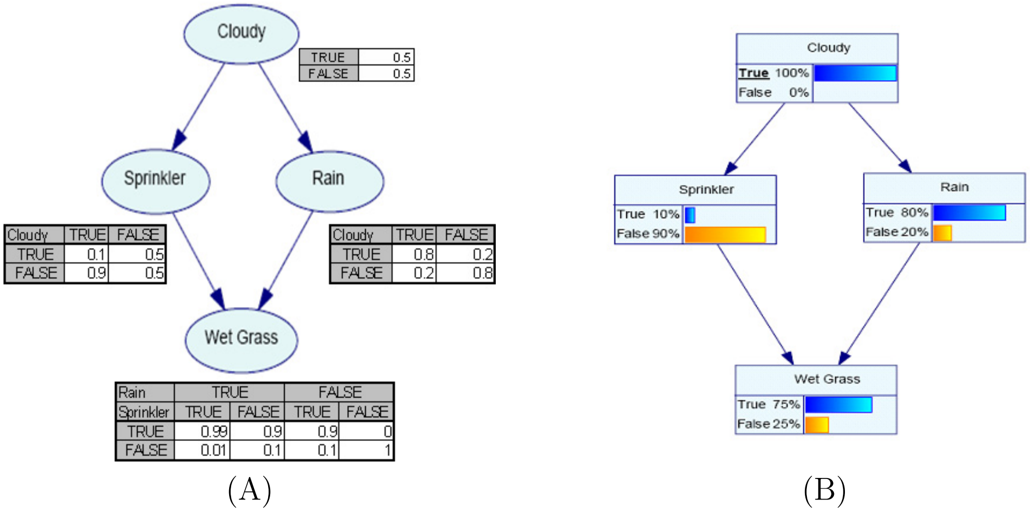


Figure 2.1: Wet Grass example for Bayesian Networks. (A) The Network, (B) Inference.

As an example consider the Figure 2.1. Part (A) illustrates a simple BN with four nodes. The variable (node) “Cloudy”, meaning whether it is cloudy today, can cause rain, represented by “Rain” variable. The edge from “Cloudy” to “Rain” represents this causal relation. Also there is a “Sprinkler” variables which indicates whether a sprinkler is off or on. If the the weather is cloudy, the sprinkler is supposed to be off but it is not always the case. This relation is captured in the edge from “Cloudy” to “Sprinkler”. Finally, we have a “Wet Grass” variable which indicates whether the grass is wet. Either a rain or the sprinkler can cause the grass to become wet, a causal relation captured in the edges to the “Wet Grass” variable. In the figure, we also see the CPT tables for each variable. For example, the CPT for “Wet Grass” node indicates that if the ”Sprinkler” is off and it is “Rain”ing, there is a 90% chance that the grass is wet.

The nodes of a BN can be either evidence or latent variables. An evidence variable is a variable of which we know its values (*i.e.*, it is measured). Bayesian Networks can be used to answer probabilistic queries. Most importantly, based on the evidence (observed) variables, the posterior probability distributions of some other variables can be computed. This process is called probabilistic inference and is the main way of using a BN once it is built. Part (B) of the Figure 2.1 illustrates an example of inference in the wet grass example. If we observe that it is cloudy today, we can use the BN to infer that there is 80% chance of a rain and only 10% chance of the sprinkler to be on. Following the Bayes theorem, and considering that Rain and Sprinkler variables are conditionally independent, gives us the following:

$$
\begin{aligned}
P(WetGrass) &= P(WetGrass|Rain, Sprinkler) * P(Rain, Sprinkler) \\
&\quad +P(WetGrass|\neg Rain, Sprinkler) * P(\neg Rain, Sprinkler) \\
&\quad +P(WetGrass|Rain, \neg Sprinkler) * P(Rain, \neg Sprinkler) \\
&\quad +P(WetGrass|\neg Rain, \neg Sprinkler) * P(\neg Rain, \neg Sprinkler) \\
&= 0.99 * 0.8 * 0.1 + 0.9 * 0.2 * 0.1 + 0.9 * 0.8 * 0.9 + 0 * 0.2 * 0.9 \\
&= 0.75
\end{aligned}
$$

The above simple computation shows the basic idea of inference in a BN; however, the inference problem can get very hard in complex networks. The computation we just saw is an example of the so-called forward (causal) inference, an inference in which the observed variables are parent of the query nodes.The inference could be done backwards (diagnostic), from symptoms to causes. The inference algorithms typically perform both type of inference to propagate the probabilities from observed variables to the query variables. Researchers have studied the inference problem in depth [29]. It is known that in general case the problem is NP-hard [28]. Therefore, researchers have sought different algorithms that perform better for special cases. For example, if the network is a polytree[2], inference algorithms exist that run in linear time with the size of the network. Also approximate algorithms have been proposed which use iterative sampling to estimate the probabilities. The sampling algorithms sometimes run faster but do not give the exact right answer [31]. Their accuracy is dependent on the number of samples and iterations, a factor which in turn increases the running-time.

Designing a BN model is not a trivial task. There are two facets to modeling a BN, designing the structure and computing the parameters. Regarding the first issue, the first step is to identify the variables involved in the system. Then, the included and excluded edges should be determined. Here, the notions of conditional independence and casual relation [70] can be of great help. It is important to make sure that conditionally independent variables are not connected to each other. One way to achieve that is to design based on causal relation: an edge from a node to another is added if and only if the former is a cause for the latter. For computing the parameters, expert knowledge, probabilistic estimations, and statistical learning can be used. The learning approach has gained much attention in the literature due to its automatic nature. Here, learning means using an observed history of variable values to automatically build the model (either parameters or the structure). There are numerous algorithms proposed to learn a BN based on history data, some of which are presented in [62].

One situation faced frequently when designing a BN is that one knows the conditional distribution of a variable given each of its parents separately, but does not have its distribution conditioned on all parents. In these situations, *Noisy-OR* assumption can be helpful. The Noisy-OR assumption gives the interaction

[2] a graph with at most one undirected path between any two vertices.

between the parents and the child a causal interpretation and assumes that all causes (parents) are independent of each other in terms of their influence on the child [126].

2.4 Summary

This chapter has presented three different lines of research that relate to the subject of this thesis. First, the literature of regression testing, as the problem subject, has been surveyed. The survey have revealed that techniques of regression testing are heavily dependent on software measurement. Specially, three areas of software measurements have been identified to be of particular relevance to the regression testing problem: change analysis, quality measurement, and test coverage measurement. The second section of the chapter has looked into each of these three areas in further details, always keeping regression testing problem in sight. As the name of the thesis suggests, the proposed framework is based on a probabilistic model. As background information, an overview of the probabilistic models used in this research, specially Bayesian Networks has been given.

The later parts of this dissertation use the provided background information in this chapter in different ways. The arguments made to justify the framework relies on the presented literature survey. The framework that is presented in the next chapter is heavily based on the measurement techniques and also the Bayesian Networks described here.

Chapter 3

The BN-based Framework

Regression test optimization, as described in Section 1.1, often concerns the fault detection power of the resulting test suites. Test case prioritization, for instance, usually is performed to speed up the fault detection. Since regression test optimization precedes regression test execution, fault-revealing capabilities of test cases are not yet known at this phase. Therefore, one can not optimize the test cases based on whether they would find bugs. However, other characteristics of a test suite heuristically correspond to its fault-revealing power. One could devise heuristics exploiting those characteristics to optimize test suites for better fault detection.

Any heuristic regarding fault detection estimates how likely test cases are to find faults. In other words, a heuristic typically uses some sort of available information to estimate the probability of a test case finding faults. For example, coverage-based approaches to test case prioritization (*e.g.,* [110] [1]) use the achieved coverage of a test case (available from a previous test execution) to estimate its probability of revealing a fault. They hypothesize (form the heuristic) that the more coverage a test case provides, the more likely it is to reveal faults; therefore, the probability of a test case finding bugs is correlated to its achieved coverage. Based on this heuristic, they prioritize test cases according to their code coverage. Estimating the probability of test cases finding faults is implicit in most existing approaches. How to estimate such probabilities (what heuristic, using what information) is indeed what differentiates the proposed techniques.

This research work aims to address the regression test optimization problem from an explicitly probabilistic point of view. The proposed framework explicitly captures fault-related heuristics into a probabilistic model and makes predictions[2] accordingly. Many software measurements provide heuristic data regarding the fault-revealing probability of test cases. The proposed framework in this chapter uses a probabilistic modelling tool, namely a Bayesian Network (BN), to relate

[1] More information provided in Section 2.1.2.

[2] Despite some technical differences, in this context, terms *prediction* and *estimation* are used interchangeably. Fault-revealing probabilities of test case are *estimated*; their outcome *predicted.*

such data and draw probabilistic conclusions as to how likely a test case is to reveal faults. As such, the proposed approach is named a *BN-based* framework.

The problem of test case optimization is broad. Finding the probability that each test case reveals faults can be used to solve many, though not all, instances of this problem. Most naturally, such probabilities can prioritize test cases for faster fault detection: test cases can be simply ordered based on their estimated probabilities of fault detection. Or, probabilities could be used to select a subset of test cases of a predetermined size (*i.e.,* size-constrained test case selection). Alternatively, one can consider a threshold on the probability figures and select all test cases above that threshold. However, note that not all instances of regression test optimization problem are solvable using this framework. Most notably, safe test case selection (or any selection that guarantees some coverage criteria)[3] could not be implemented using probabilities. Relying on the probabilities, the proposed approach (at least at its current stage) does not provide any means of guaranteeing properties related to the resulting test suites.

Proposed BN-based framework seeks the following research and practical goals:

- Increasing the precision of the optimization through the use of multiple heuristics and measurements. Since introduction of faults to the system is a probabilistic process (*i.e.,* it could not be fully modelled and predicted) any regression technique is based on heuristic measures. Each heuristic uses some available data as *indicators* of where faults might reside to predict the fault revealing abilities of test cases. As such, aggregating many heuristics should be fruitful. A multi-heuristic solution is likely to perform better and more consistency because in predicting a probabilistic phenomena, combining different correlated data can help mitigate the shortcomings of any single indicators. The proposed framework aims to facilitate the logical incorporation of different heuristics into one model and hence provides a more sophisticated estimation regarding fault revealing capabilities of test cases.

- Providing flexibility in terms of which heuristics and measurements to use. Existing studies in software measurement have established that the fault revealing power of different measures vary across software projects. Moreover, not all measurements are available for reasonable cost in all environments. An approach that provides flexibility in terms of input measurements can help test engineers adapt their regression test optimization techniques to the realities of their environment based on their intuition or empirical experience. This framework aims to provide flexibility in measurement techniques.

The following section provides an overview of the proposed BN-based framework in a general sense. A later section elaborates on a process to realize and implement the general framework.

[3] More information presented in Section 2.1.1

3.1 Overview

The proposed framework could be thought of as a general approach to the optimization problem that needs to be realized before being applied to a particular environment. This section presents an overview of the underlying ideas behind the framework and how it generally works. How to implement these ideas to get realizations for specific environments follow in later sections.

Recall that proposed framework addresses the problem of regression test optimization for better fault detection from a directly probabilistic point of view. It seeks to estimate the probability that each test case reveals a fault if executed in the regression testing phase. When the goal of optimization is to increase fault detection power (*i.e.,* rate), such estimations could be directly used to prioritize or select test cases. To predict the outcome of test cases, like any other prediction, one needs to first model the events that lead to the event in question and then use available measurements from the system to come up with numerical estimations[4].

3.1.1 Events and Variables

The *regression test system* can be characterized in three sets of events:

(i) **Artifact Change:** everything starts with a change in a software artifact that has been previously verified and/or validated.

(ii) **Fault Introduction**: those changes may lead to the introduction of regression faults in the changed or change-impacted artifacts.

(iii) **Regression Test Execution**: testers re-execute existing test cases that attempt to reveal introduced faults.

These three categorizes of events capture, essentially, the entire scenario that affects the outcome of a regression test execution. Clearly, this characterization is a general one and, in specific environments, finer-grained events could be identified. Nonetheless, having these categories enables us to model the system. The uncertain nature of these events (as reflected in the use of *may*) encourages their modelling using probabilistic tools. In probabilistic modelling, events are captured using random variables.

Given a set of software artifacts of interest $\{a_1 \ldots a_i \ldots a_n\}$[5] and also a test suite $< t_1 \ldots t_j \ldots t_m >$, the above events are modelled through three sets of random variables:

[4] For a discussion on measurements in software prediction the interested reader may refer to [48]

[5] A software artifact, here, can refer to any entity in software development life-cycle and in any level of abstraction. It can refer to a use case in requirements documents, a component in a design documents, or a structure in source code (such as a class, method, or statement), or even an element of byte code (such as a basic block). Any realization of this general framework needs to specify the exact meaning of an artifact

(i) C **Variables:** change in any artifact a_i is a random variable C_i with two possible values: *changed* and *unchanged*.

(ii) F **Variables**: the existence of a regression fault in an artifact of interest a_i is a random variable F_i with two possible values: *faulty* and *sound*.

(iii) T **Variables**: the result of executing a test case t_j is a random variable T_j with two possible value: *failed* and *passed*. [6]

Associated with each random variables is a probability distribution $P(X)$ that indicates our estimate of the likelihood of each of the two possible outcomes. Of these three sets of variables, the distribution of some can be observed (*i.e.,* can be measured from the software system). Change variables, for example, could be measured by looking at consecutive versions of software and identifying changes. Distribution of others, however, such as F variables could not be directly measured. What we are trying to find is the distributions of the last set of variables $P(T_j)$s. G variables are also not observed variables because before running test cases we can not be assure of their value.

3.1.2 Variable Interactions

Looking at the probability distribution of variables in isolation ignores the interactions between them. Looking back at the description of the three sets of events, it is evident that there is an interaction between variables: a change plausibly leads to introduction of faults and faults plausibly lead to the failure of test cases. Therefore, from a probability theory point of view, these events are not all independent from each other. Provided a full system of conditional distribution functions, where each variable is conditioned on the combination of all other variables, one can theoretically infer a posterior distribution for unknown (query) variables based on observed variables using Bayes' theorem (as described in Section 2.3). Such a task of *probabilistic inference* is computationally complex and infeasible in systems of large size. Modelling such a system, moreover, requires distribution functions conditioned on all the combinations of all other variables. This makes the task of modelling intractable.

In real systems, however,although most variables are dependent, many are *conditionally* independent, meaning that knowing the distribution of a third variable makes their outcome independent. The notion of conditional independence, although different, is related to causal relations: if A is not caused by B, and if

[6] All the events are modelled as binary variables with two possible outcomes. This does not necessarily need to be the case in modelling the described system of events. Change variables, for example, could have three values of changed, unchanged, and change-impacted. Binary variables are used here because they are easier to present and also simpler to model. The use of binary variables allows us to think of them as true/false variables with the first mentioned value as true and the other as false. For example, $P(C_i = \textit{changed})$ could be simply denoted by $P(c_i)$ and $P(C_i = \textit{unchanged})$ by $P(\neg c_i)$. Such a notation is used throughout this work.

its real cause (C) in known, A could be designed to be conditionally independent of B because knowing the distribution of C makes the outcome of A independent of B. Knowing causal relations, one can assume variables that do not cause each other directly are conditionally independent to simplify the system both in terms of modelling and inference. Essentially, one needs to state the distributions of a variable conditioned merely on those on which it is conditionally dependent. In other words, conditioning a variable on those from which it is conditionally independent is not necessary. By modelling lack of causal relation as conditional independency, a variable needs to be conditioned only on its causes. This property is what makes the notions of causal relations of interest.

In the system described above, we assume that the following are the only causal relations.

(i) Faults (F variables) may be caused by changes (C variables).

(ii) The failure of test cases (T variables) may be caused by the existence of faults (F variables).

Therefore, in the model proposed by this framework an F variable is conditioned only on C variables and T variables are conditioned merely on F variables. That is the model assumes that T variables are conditionally independent of C variables; a property that makes sense because knowing the outcome of F variables (whether software is faulty) makes the outcome of C variables (whether software has changed) unimportant. Based on these assertions, three sets of distribution functions are required in order to complete the modelling of the system: $P(C)$, $P(F|C)$ and $P(T|F)$.

3.1.3 Distribution Functions

Distribution functions should be based on numerical values that reflect the system in question. In practice, one needs to look at the history or properties of a system in order to estimate probability distributions. Software measurement techniques provide us with numerous ways of extracting information regarding the characteristics and the history of software. Such information could be used to estimate the required probability distributions. For the three required set of distributions ($P(C)$, $P(F|C)$ and $P(T|F)$), the following measurements are required respectively:

- identifying whether a software artifact has changed since the previous version.
- estimating the probability that an artifact becomes erroneous as a result of a change in the system.
- assessing the probability of a test case finding a fault residing in an artifact it exercises.

These items each correspond to a well-studied topic in software measurement area. Identifying changes is the subject of *change analysis*. The probability of fault introduction to an artifact depends on its quality. Mathematical models exist that utilize metrics of quality to estimate the fault-proneness of software artifacts. The probability of a test case finding a fault contained in an artifact could be estimated by the coverage that it provides on that artifact. These three areas of software measurements are explained in Section 2.2. As described there, a variety of measurement techniques and metrics have been developed for each of these three categories. Experiments have revealed that the prediction power of these techniques vary across different projects [10]. To get more accurate estimates, therefore, one needs to choose metrics based on the specific environment where the framework is used. Engineers can choose specific metrics based on their experience and knowledge of the system in question. Empirical studies on the history of the project (or similar projects) can also help selecting suitable metrics.

Once the specific measurements to be used are chosen, the corresponding metrics need to be calculated from the software system. Based on calculated numbers, then, the required probability distributions can be calculated. That could be done by assuming a functional form and estimating the parameters using available data. Alternatively, one can simply manipulate (*e.g.,* normalize) the gathered numbers to estimate probability values that form the distributions.

To sum up, we saw how the proposed framework first models the system using tree sets of random variables. Then, we saw how some of these variables can cause the others and how assumptions of conditional dependency can be use to model them. Next, we saw that the distribution of the variables conditioned on the other variables can be estimated using software measurement techniques. Now, for the proposed framework to achieve its goal, a modelling tool is needed that can put all these data together. Such a tool should also provide a means of probabilistically inferring the distribution of T variables using all the gathered information.

3.1.4 General BN Design

As described in Section 2.3.1, a Bayesian Network is a tool to capture conditional independence among variables and to facilitate probabilistic inference. These characteristics make BNs the exact tool needed here. It is the natural choice to model the described system given the aforementioned summary of what this framework intends to accomplish. Bayesian Networks are suitable modelling tools here because:

- They provide a means of probabilistic modelling of a system. Particularly, they allow modelling causal relations in a system as conditional dependencies and explicating conditional independencies. This, in turn, facilitates incorporating many sources of information into a single model based on logical relations between them (causal relations) rather than ad-hoc gluing.

- They enable inferring the posterior probability distributions of random variables of interest, given any information about other variables of the system. The literature of BN is rich with algorithms of probabilistic inference developed for networks with different properties.

- They are graphical models that could be easily understood and manipulated by engineers if necessary. Moreover, a handful of tools exist to help build, understand, and modify BN models.

This BN-based framework, as the name suggests, uses Bayesian Networks to put all the pieces of the puzzle together. A BN, a DAG of random variables, is comprised of nodes, arcs, and Conditional Probability Tables (CPT). Nodes represent random variables, arcs indicate conditional dependencies among those variables, and CPTs are tabular representations of distribution functions. These three elements are designed as follows.

Nodes: All identified random variable (C_is, F_is, and T_js) are represented each as a node. Corresponding to each artifact a_i in the system are two nodes (a C_i and a F_i) and to each test case t_j is one node (T_j).

Edges: Based on identified causal relations, two sets of edges should be present in the model: $C_i \rightarrow F_k$ (CF edges) and $F_i \rightarrow T_j$ (FT edges).

CF : A change in artifact a_i can cause faults in a_i or those artifacts impacted by it. Therefore, CF edges are for sure needed from a C_i node to the corresponding F_i. In addition, if another artifact a_k is likely to be impacted by changes in a_i, a CF edge could be made from C_i to F_k. This consideration of change impacts complicates the model significantly both from modelling and computation perspectives. A particular implementation of the framework can choose to account for such effects or not.

FT : A test case t_j can reveal a fault in an artifact a_j only if it exercises that artifact. Therefore, FT edges are required from an F_i node to a T_j only if t_j covers a_i.

CPTs Associated with each set of random variables is a CPT.

$CPT(C)$: Having no parent nodes[7], these CPTs include only one cell holding the $P(c_i)$ value. The probability of change in an artifact is observable by differencing consecutive versions. This value is typically either 0 or 1 (either changed or not). As a trick, one could use the extent of change to come up with a probability number in [0,1] interval. In that case, this figure should be interpreted as the probability of *effective* change.

[7] A node A is said to be the parent of B if there is an edge from A to B

$CPT(F|C)$: The values in this table indicate the probability of fault introduction to an artifact. As mentioned before, models of fault-proneness should be used to estimate these probabilities. Those models use measures of quality to associate with each artifact a relative number indicating its fault-proneness. Those numbers can be normalized to get probability values for cells of this CPT. In case of including change impact edges, effects of other artifacts should be also measured (through change impact analysis) and incorporated into this table.

$CPT(T|F)$: Since a T variable can be conditioned on many F variables, these CPTs can potentially have numerous cells. The CPT of a T variable should, in the general form, have probability values for all the combinations of the outcomes of parent F variables. For example, if a test case is covering five artifacts, its distribution is conditioned on five binary variables and as such 32 different probability values should be specified. Estimating all these values directly is a daunting task. Using the Noisy-OR assumption (described in Section 2.3.1), one can reduce the table to five numbers, each for estimating the effect of one F variable on the T variable. Generally, Noisy-OR allows specifying the effect of each cause on a variable separately. The assumption, informally, is that the event of (not) revealing faults by a test case in one artifact is independent from those of other artifacts. Based on the characteristics of the test suite this might be a reasonable assumption. Nonetheless, here this assumption is made because of the simplicity it provides. With this assumption, we only need the probability values for each test case finding faults in each artifact regardless of the state of the others. Such a probability could be easily estimated by the coverage that a test case provides on that artifact. The Noisy-OR assumption, then, combines all these separate estimates to get one CPT.

Once a BN is built to model the software system, it could be used to infer the probability distribution of T variables.

3.1.5 Probabilistic Inference

Algorithms of probabilistic inference can be used to automatically compute the posterior distribution of variables given a BN model and any observed variable. These algorithms, in essence, propagate the probability distributions from known variables to unknowns. Bayes' Theorem is used to derive such a propagation. In the case of the proposed BN, distributions are propagated from C to F variables and then from F to T nodes. A resulting $P(t_i)$ number indicates the probability of a test case t_i finding faults. Such a number, in short, is based on the coverage that t_i provides on the artifacts that are fault-prone and also changed or change-impacted. These numbers are useful for regression test optimization. An optimization algorithm is needed to use these numbers for building the final optimized test suite.

In case of size-constrained test case selection, where a predetermined number of test cases need to be selected, one optimization algorithm is to simply choose test cases that have higher probabilities. Another algorithm, as Kim *et al.* suggests [73], is to select test cases randomly based on their probability distributions. This can help finding the residual faults. For prioritization, sorting test cases based on their probability of finding faults is the simplest algorithm of optimization.

Just as feedback is used in code-coverage-based prioritization techniques (as described in Section 2.1.2), it can also be used here. A greedy optimization algorithm can incorporate feedback by adding test cases to the prioritized order in iterations. In each iteration, first, the probabilistic inference is performed; then, one or more test cases that have higher probabilities of revealing faults are added to the prioritized order; finally, those test cases are marked as observed variables with $P(t_i) = 0$. The iterations are halted when all the test cases have been added to the final order. This greedy framework has the advantage that the effects of adding each test case to the prioritized order are taken into account for the next selection. This algorithm is used here for optimizing test cases[8].

The provided overview have clarified the ideas behind the proposed framework. It has been described in high levels of abstraction and for a general regression test environment. Mathematical basis of the approach and the reasons behind using different tools and measures have also been presented. To apply this framework, one needs to develop a implementation of it. Such an implementation would be based on specific choices of software artifacts, specific levels of granularity, specific measurement techniques, etc. The next chapter of this dissertation presents one implementation of the framework which is applicable in a wide range of systems. Before that, the rest of this chapter presents a general process model for implementing the framework.

3.2 The Process Model

The presented BN-based framework is a general one which needs realization (or implementation) before it could be used. A realization needs to give many details that have been intentionally left unspecified. To enable an implementation, one needs to decide on factors such as: the exact definition of an *artifact*, exact measurements for probability estimations, and the exact topology of the BN.

Figure 3.1 depicts the process of implementing a realization. The diagram depicts two different processes. The first process is that of *BN-based Framework Realization*. The result of this process is an implementation of the BN-based framework in the form of a *regression optimization tool*. This tool is used in the second part of the figure, *regression testing process*, by software test engineers in order to plan the regression test phase. After planning, testers execute the created regression test suite and evaluate the results. The knowledge gained from test execution, in turn,

[8] The details of the algorithm are presented in Section 3.2.

could be potentially used to recalibrate the implementation. *Calibration* of the tool basically means repeating the process of realization with the new knowledge about the prediction power of metrics in hand. As a result of the new knowledge, different metrics and design decisions could be made and the final tool can change.

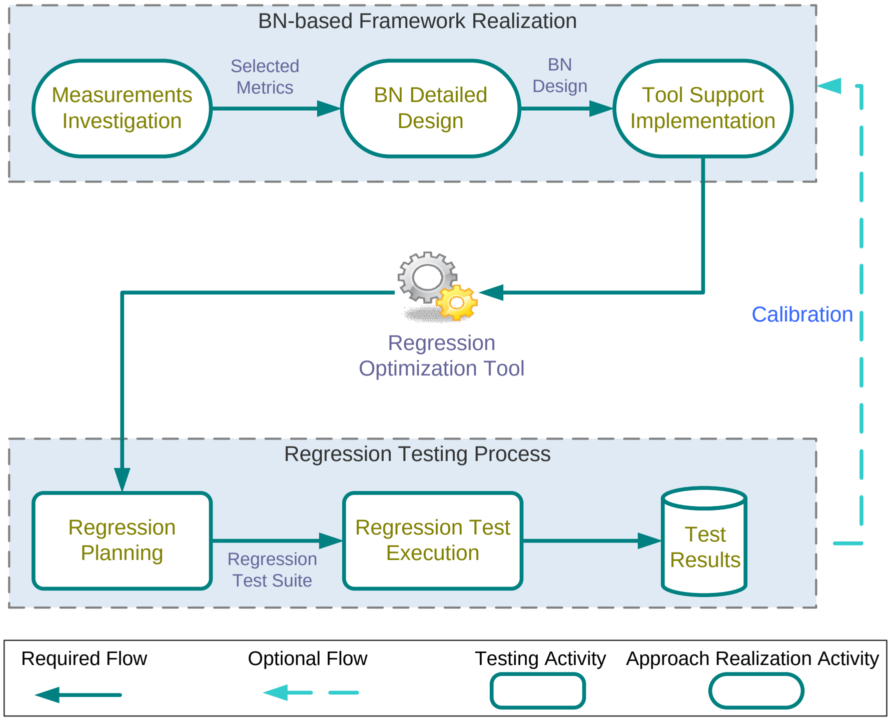

BN-based Framework Realization
Measurements Investigation
Selected Metrics
BN Detailed Design
BN Design
Tool Support Implementation
Regression Optimization Tool
Calibration
Regression Testing Process
Regression Planning
Regression Test Suite
Regression Test Execution
Test Results
Required Flow
Optional Flow
Testing Activity
Approach Realization Activity

Figure 3.1: The Process of Realizing the BN-based Framework

The process of realization, the focus of this section, is contains three main tasks. First, a phase of *measurement investigation* is required in order to decide what artifacts and metrics should be included in the model. Then, based on the results of the investigation, one needs to settle on a *BN detailed design* which shows how exactly a BN should be built based on available data (*i.e.*, metrics). Based on that design, *tool support implementation* is done to get the *regression optimization tool* that realizes the framework. Each of these steps is further explained in the following sections.

3.2.1 Measurement Investigation

As explained, a realization needs to specify what an artifact is and what metrics are being used. These two questions are related and must be answered together.

The term *artifact* here can refer to any entity in software development life-cycle and in any level of abstraction. A use case in requirements documents, a component in design documents, a structure in source code (such as a class, method, or statement) or byte code (such as a basic block) are all example of what can be defined to be an artifact. This flexibility is provided because in different environments, different information are available to testers. Even when all information available, dealing with some artifacts can be simply less costly than others. For example, in the processes deploying Requirement-based Testing (RBT), information regarding requirements are easier to gather. Or, in large distributed heterogenous systems, dealing with source/byte code directly can be very costly.

In choosing what an artifact is, one needs to make sure that the three sets of required measurements can be cost-effectively gathered. Measuring change in the artifact should be possible, its quality should be quantifiable, and the coverage of test cases on it should be traceable. When information available about different artifacts, then those that could be more accurately measured are a better choice. Lower level artifacts (such as code related structures) are naturally easier to measure precisely than higher level ones (such as requirements). As a rule of thumb, the most low level artifacts for which required measurements are meaningful and also cost-effectively available are a good choice.

Even with an exact definition of artifacts, many different metrics could be still available for each of the three sets of required measurements. Changes in byte code, for example, could be measured through a handful of algorithms (some described in Section 2.2.1). In fault-proneness measurement even more options are available. The notion of quality is inherently vague and hence measuring it is a daunting task; an abundance of different techniques and metrics are proposed for this purpose. In this step, we need to also choose among these alternatives.

Considering that measurement in this framework is a means of probability estimations, in an ideal situation, the choice of metrics would be based on empirical data regarding the prediction power of a metric. If data are available from previous projects or releases regarding where faults resided, what test cases detected them, and so on, one can use mathematical tools to compare the prediction power of measures. Statistical methods of computing correlation and regression analysis [25] are examples of such mathematical tools. In reality, however, these data are not available most of the time (engineers do not have enough motivation to gather and analyze such data). Therefore, the choice of metrics need to be made based on the intuition of testers and the results of existing research regarding prediction power of the metrics. An overview of measurement techniques is presented in Section 2.2.

3.2.2 BN Detailed Design

A generic BN design has been introduced in Section 3.1.4. There are, however, details regarding the network topology that need to be decided separately for each implementation. Based on the definition of the artifacts and the metrics to utilize (available from the last step), the shape of the BN is clear for the most part. Still, there are flexibilities in the design of BN that should be addressed.

One issue, here, is whether CF edges are required from a C node to F nodes of other artifacts. That is, whether change impact should be considered. The investigation of available measurements from the previous step answers this question partially; for change impact edges to be included, one needs to have measures of change impact. If such measures are available, one may still decide to discount them because of the computational complexity that they impose on the inference.

Another issue is how the gathered measures should be represented in the CPT (tables) of the BN. Collected metrics are typically not probability values but rather relative numbers that show the strength of relations. One needs to derive probability distributions from such numbers. One way of achieving that is through normalizing the value. Finally, note that if the goal of optimization is merely prioritization, the method of deriving probabilities is less important because the relative position of test cases is not significantly impacted. The absolute probabilities, on the other hand, are affected and hence if those numbers matter for the goal of optimization, probability derivation should be performed carefully, by using historical data for example.

3.2.3 Tool Support Implementation

Implementing a designed BN to get the optimization tool is a relatively straight forward task. To assist such implementations, a general implementation model is presented here. This model lays out different components that comprise the framework. Specific realizations differ in how they exactly implement these components.

A schema of a generic implementation is illustrated in Figure 3.2. There are three main steps involved as follows.

Collecting Measures applies measurement techniques to gather the data that is to be included in the model. Three main sets of data need to be collected and correspondingly three sub-steps are required: *change analysis*, *fault-proneness modelling*, *test coverage calculation*. Typically, existing tools are used in order to make these measurements. Also, note that theses measurements are not necessarily automatically generated. In some cases (for example when informal requirements are the underlying artifacts) required metrics could be estimated by involved people and based on their intuition.

Building the BN creates an actual BN for modelling the system based on the gathered measurement data. Building a BN involves two tasks of *Creating the Graph Structure* and *Computing the CPTs*. From framework realization, a particular design for the BN is available. That design details the topology of the graph to be built and also the method of deriving of CPTs. Based on that detailed design, this step automatically builds a BN modelling the whole system. Creating, manipulating, and using BNs are by nature hard tasks that need to account for many mathematical details. Fortunately, there exist a handful of off-the-shelf tools that facilitate these tasks (a comprehensive list is provided in Appendix B of [76]). Some tools provide APIs to work with; using those tools enables automatic creation

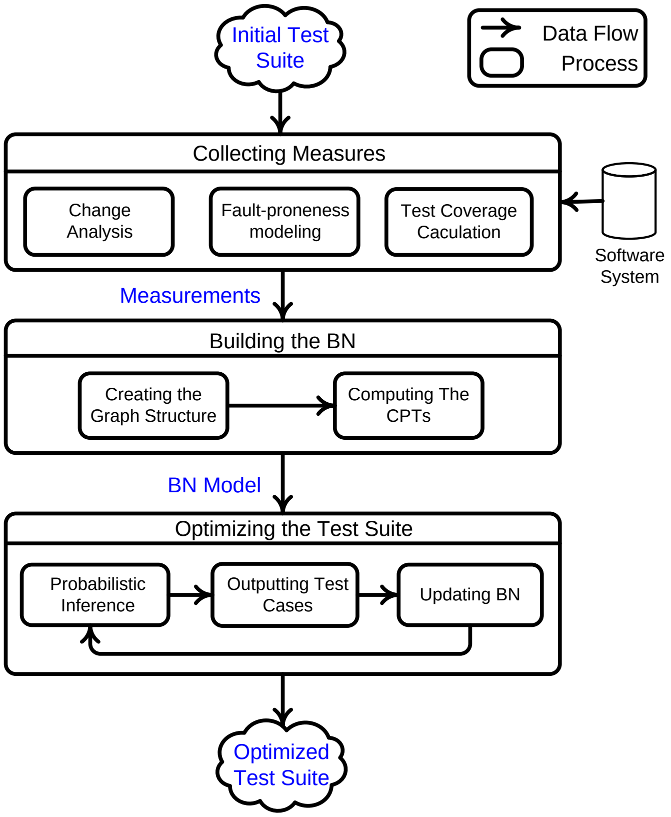


Figure 3.2: A Generic Implementation for the BN-based Framework

of the model. These tools also provide means of persisting a BN using special XML-based formats[9]. Such tools could be used, here, to manipulate the BN.

Optimizing the Test suite is the last step, where the output is built. In this step, an optimized (*i.e.,* prioritized) test suite is generated based on the goals of regression task. A feedback-employing greedy algorithm of optimization is used here in three sub-steps. First, *Probabilistic Inference* computes $P(t_i)$ values using the constructed BN. Next, *Outputting Test Cases* adds some test cases to the output test suite based on computed $P(t_i)$s. Finally, *Updating the BN* uses outputted test cases to modify the model to account for the effects of added test cases. This modified BN is then fed back to the *Probabilistic Inference* step and the loop continues until the output *optimized test suite* is constructed. Each of these steps involves interesting issues:

- *Probabilistic Inference*: An algorithm of probabilistic inference should be used here to get $P(t_i)$ values. Algorithms of probabilistic inference are well studied in the literature of BNs [101]. Since the problem is NP-Hard in general, numerous algorithms have been proposed, each suitable for certain situations. Simple experiments can be performed to find the algorithm most suitable for

[9] Genie/Smile is an example of tools that provide all these features [52].

an specific design of the BN. Because the graphs of the BNs are not polytrees[10] here, any algorithm would have an exponential worst case performance.

- *Outputting Test Cases*: Based on the probabilities from the previous step, a number of test cases are selected and added to the output test suite. A parameter called *stp* controls the number of added test cases. In one extreme, this parameter could be set to the size of the entire test suite. In this case, all test cases would be added at once and hence there would be no feedback mechanism. In the other extreme, we can set this parameter to be equal to 1; in this case, feedback is used in its full capacity. Any number in this range can be used to have a feedback mechanism in place, while performing the computation intensive task of probabilistic inference less often. This parameter provides the flexibility to adjust the running time.

- *Updating BN*: In this step, the variables corresponding to the test cases just added to the output are marked as *passed* evidence nodes (*i.e.,* $P(t_i) = 0$). As a result, the probability of being faulty for the artifacts covered by those test cases are reduced. Consequently, the other test cases that test the same artifacts will have a decrease in their probability of failure. Note that regardless of whether a test case fails or not, we mark its variable as a *passed*. There are two reasons for this. First, during optimization it is not know yet whether the test case is going to fail. Second, if we mark a node as *failed*, then the other test cases that cover the same elements will have an increase in their probability of success. This is not desirable because we have already covered those elements at least partially. Therefore, we simply mark outputted test cases as *passed*.

In case of prioritization, optimizing test cases finishes when all test cases are added to the test suite; in case of selection, when enough test cases are selected.

3.3 Summary

To summarize, a general BN-based framework to solve the test case optimization has been presented in this chapter. An overview of the framework, ideas behind it, and its mathematical basis have started the chapter. Then, a process model has followed showing how it should be implemented based on specifics of an environment to get an regression optimization tool. In doing so, a general implementation of the framework has been presented.

This general framework needs to be realized first in order to get a working system. Most research in the area of regression testing is performed on software for which code is available. The next chapter presents a specific code-based implementation of this framework.

[10] a graph with at most one undirected path between any two vertices.

Chapter 4

A Code-based Realization

The BN-based framework, presented in the previous chapter, is general by design. The claim is that different systems must be treated differently, mainly due to the dissimilarities in suitable sources of information (*i.e.,* measurement techniques). The proposed framework needs to be realized for a specific software system (or families of systems) before it could be implemented. A realization is applicable whenever the required sets of measurements are available.

This chapter presents a code-based realization of the proposed BN-based framework. In the rest of the chapter, first the motivation behind selecting such a realization is described. Then, based on the process model of Section 3.2, the details of this particular realization are elaborated. Each subsequent section discusses one step from the aforementioned process.

4.1 Motivation

There are many existing approaches in the regression testing area that, like ours, utilize a set of measurements to heuristically predict the fault-detection power of test suites. These existing approaches mostly depend on particular sets of metrics for their tasks. Therefore, it could be said that they implicitly assume the availability of those metrics. Availability, here, refers to the existence of artifacts upon which a specific metric is defined and also the ability to collect the corresponding measures from a system. Requirements-related measures, for example, are not always available because they can not be collected on many software projects in which requirements are not fully managed. An approach is applicable only when the underlying artifacts are available. As expected, researchers have sought to build up their approaches on the most available set of artifacts.

Most of the existing approaches are *code-based* in the sense that the metrics they use are defined based on code-related artifacts. This renders developed approaches applicable to most software systems because any working software (that is to be tested) certainly has its code. Relying on the code, in other words, is the safest assumption. Although code-related measurement could be overwhelmingly costly or imperfect in modeling particular systems, it is nevertheless the most *applicable* approach. Furthermore, software measurement techniques are unsurprisingly much richer on software code compared to less formal artifacts such as requirements or design[1]. This provides the code-based techniques with a variety of relatively precise measurement techniques to exploit. Code-related metrics such as *code* coverage, *code* fault-proneness, and *code* change have all been used in existing regression testing techniques (details provided in Section 2.1). In addition, there are difficulties in the technicalities of research on artifacts other than code that has further inclined researchers towards code-based approaches. Researchers need to rely for their experiments mostly on open source and publicly available software, for which many of the life cycle artifacts do not even exist. Other artifacts are mostly built and maintained in industry-scale projects but access to such data is limited due to confidentiality measures taken by most organizations.

A code-based realization of the proposed framework facilitates the comparison with other research work in this area. Moreover, it can take advantage of the aforementioned properties of a code-based technique such as access to a handful of well-studied measurement techniques. It also is the most practical approach from a research point of view [2].

The process for the realization of the BN-based framework has been presented in Section 3.2. According to that process, three main steps of *measurement investigation*, *BN detailed design*, and *tool support implementation* need to be taken for realizing the framework. To describe this particular code-based realization, this chapter takes the same route and follow the prescribed process. The followings sections each correspond to one of the steps in the realization process.

4.2 Measurements Investigation

The first step towards the realization is investigating available measurement techniques that can provide the three sets of required data. In this step, the artifacts to be modelled and the exact metrics of prediction on those artifacts need to be selected.

The availability of the code-related artifacts has been discussed as a motivation for this code-based realization. Code not only is a relatively available artifact, but also is the richest source of information about software. Three set of measurements are required in the BN-based framework, all of which could be well measured on code as the underlying artifact.

[1] The emergence of new software technologies, such as model-based development, creates new artifacts that are formal enough to allow precise measurements. Research in regression testing has already started to take advantage of such artifacts (*e.g.,* [99]).

[2] It should be noted here that other realizations that rely on other artifacts have been developed for industrial environments. This dissertation does not report on that line of research mainly due to confidentiality issues.

- **Change:** The most reliable source of change information is from the code itself. Even if different artifacts are created for a software project, they are not guaranteed to be maintained with the program. Software changes are likely to be made to the code without being reflected in any other artifacts. Furthermore, analyzing change in code is fairly easy (through methods described in Section 2.2.1), mostly because of its formality.

- **Fault-proneness:** Although fault-proneness is hard to predict in any artifact of software, there exist models that use structural properties of the code (such as complexity, coupling, and cohesion) to predict its fault-proneness. Although far from perfect indicators, the results from these model do correlate with fault locations (as discussed in Section 2.2.2).

- **Coverage:** Coverage is typically traceable on the code. As described in Section 2.2.3, source/byte code of programs can be instrumented with probes that report execution paths. When running test cases initially (*i.e.,* for the previous version), such reports could be generated; from those reports, code coverage data can be computed.

The availability of the required measurements for the code-related artifacts supports the idea of using them as the artifact. The notion of software code, itself, is a broad one. Code can refer to source code or any of its machine readable variations commonly called execution or byte code. Each of the two, in turn, could be looked at in different levels of granularity. On the source code, structures such as statements, methods, modules, classes, packages, and components are available. On the byte code, in addition to all the source code level artifacts that are also tractable in byte code, there exist other structures such as basic blocks. The choice between these levels of granularity should consider both the availability of required data and performance issues.

There is a performance accuracy trade-off in the choice of the level of granularity. Moving to more fine-grained structures such as statements potentially provides more accurate information. On the other hand, it can make the pool of artifacts large and hence increase the complexity of involved computations. A typical project these days can easily contain millions of lines of code. Choosing statements as artifacts needs a modeling tool and algorithm that can handle a million entities. Inference algorithms in Bayesian Netwoks are exponential in the worst case and hence do not allow such computations. Also note that the empirical studies on the existing techniques of regression indicates that moving to higher-level structures does not necessarily decrease their performance significantly (*e.g.,* [43]).

Working on code-based artifacts makes a particular realization depend on a specific language (or a family of languages in which similar structures exist). This proposed realization of the framework focuses on the Object Oriented (OO) languages, mainly because of their prevalence in industry. Most concepts and language structures are shared among OO languages such as Java and C++ and hence it is not necessary to further narrow down the language boundaries.

In OO systems, the concept of a *class* has properties that makes it a suitable choice for the definition of artifacts. First, a class is on the right level of granularity for the type of analysis we perform. On one hand, it is high-level enough to enable computation for large systems. The number of classes in a typical system, a number in order of hundreds to thousands, is in the boundaries of available computational resources as needed by BN algorithms. On the other hand, the notion of classes are not overly high level to make the gathered information imprecise. The principles of OO design put a great emphasis on the notion of cohesion in a class. Designers are encouraged to create a class such that all of its parts are closely related and perform a single functionality. A class, therefore, could be easily thought of as one entity, as opposed to higher level constructs such as a package. These reasons have also encouraged the research on measurement techniques to define many metrics to a class-level. That adds to the value of choosing classes as the artifacts.

This realization of the framework, based on the aforementioned reasons, chooses to have classes in a OO system as the artifacts. Note that a class is represented both in the source and byte code and hence there is no need to restrict the definition to either one. The required sets of measurements are indeed defined for this level of granularity, as described in the following.

4.2.1 Change

The change data required for the purpose of this realization need to estimate "the probability of each class having changed from a previous version". The changes between two versions of a code is something that could be observed (*i.e.,* there is no uncertainty involved). One can look at the code in either source or byte code representations and use a simple string matching algorithm to decide whether each class has changed or not. In that case, each probability would be a zero or one number. The problem with such an approach is that the amount of change in a class would not be taken into account in the model.

The nature of changes in software systems vary. One may simply rename a variable using refactoring tools, a relatively safe modification, or change an implemented algorithm, a very error-prone edit. Moreover, a class may be changed altogether to introduce new features, or changed slightly to fix a simple well-understood bug. These changes are different for our purpose in that their error-proneness is different. The fault-proneness variables in our model capture how error-prone artifacts are inherently, regardless of the amount of change they have gone through. The notion of error-proneness for modification itself is different from fault-proneness of artifacts. This concept also can be incorporated to the model.

As pointed out in Section 3.1.4, $P(C_i)$ values could be used to indicate the amount of change. In this case the interpretation of the $P(C_i)$ values as the probability of change is not possible; instead, they could be interpreted as the probability of an *error-prone* change. Because we do not have a precise definition of an error-prone change, such an interpretation can not be formalized but still this trick helps incorporate useful information into the model. To estimate the error-proneness of a change, a measure of the amount of change is a possible option. The heuristic behind such a metric is that the more a class has changed, the more error-prone the changes are. This framework requires a quantification of change in each class.

To quantify the amount of change, different methods could be used. Section 2.2.1 introduced some existing works in this area. Algorithms as simple as GNU diff could be used here. More complex byte code-based measures such as `Sandmark`'s Constant Pool LCS are also available [22]. Although one can conjecture that more complex algorithms provide better results, such a claim can be evaluated only through empirical means. Later chapters of this thesis will present an empirical study that among other things investigates this question.

4.2.2 Fault-proneness

Code-based models of fault-proneness have been discussed in detail in Section 2.2.2. These models are based on the metrics of software quality. Software quality metrics typically measure a structural property of the code which is claimed to correlate with its quality. For example, it is claimed that the more coupling a class has to other classes, the more it is prone to faults. If this claim holds, which empirical studies show in many cases does, measures of coupling could be used to spot fault-prone parts of the code. One can define a measure of class coupling based on the number of other class to which a class interacts.

There are numerous quality metrics defined and claimed to be correlated with quality (esp. fault-proneness) of OO code. To establish models of fault-proneness these metrics could be also mingled together in a mathematical model, to get more elaborate predictors named multivariate models. These multi-variate models need to be first trained on other systems (for which fault locations are known) and then used on the software in question. This renders those models costly to build. Univariate models (those based merely on one metric) are easier to construct but the question still remains as to what metrics should be used. Again, empirical investigation is the best way to answer that question. Metrics from the Chidamber and Kemerer (CK) suite [21], as probably the most widely used set of metrics, are investigated in a later chapter of this thesis.

4.2.3 Coverage

Coverage is an easy to gather metric when code is the underlying artifact. Measuring the coverage on statement or basic-block levels is only a matter of code instrumentation and tracing what has been executed. To abstract such data to class level, one can simply count the number of blocks (or statements) covered in a class. Dividing those figures by the total number of blocks also gives normalized measures that indicate the percentage of coverage in each class.

The important point about code coverage information is that it needs to be gathered on a previous version of the programs. When planning the regression testing of version v, the test cases have not been yet executed on P_v. Therefore, their exact coverage is not known. Running test cases in order to gather coverage data would only defeat the purpose. The regression test suite, however, have been previously executed on the P_{v-1}. If traces from that execution are available, the coverage for P_v can be estimated. In simplest form, one can assume that the class coverage of classes remains unchanged from the last version. Clearly this assumption does not always hold. However, research has shown that the coverage of a previous version provides close estimations of that of the current version [107]. Although approaches of estimating the current coverage from the previous coverage exist [41, 115], in this realization, we choose simplicity; we approximate the coverage of a test case t on P_v by that of P_{v-1}. This approach is in line with other research in this area.

The step of measurement investigation provides for us with a basis for designing the BN. Knowing that what exactly the artifacts are and what information needs to go to the model, one can detail how the BN model should be built.

4.3 BN Detailed Design

A Bayesian Network consists of three elements: nodes, edges and CPTs. A detailed design of the BN needs to specify how each of these three elements should be exactly built. Such a design would be based on the general design presented in Section 3.1.4. For this code-based realization, and based on the observations from the measurement investigation step, the following details the specifics regarding each element of the BN. At the end, a summary of the design is given through a simple diagram.

4.3.1 Nodes

Three set of nodes exist in the proposed BN model, each corresponding to a set of random variables described in Section 3.1.1.

C Nodes:

Each class in version v of the program under analysis has a corresponding node of the type C. A C_i node, corresponding to the event of change in the class a_i, has two possible values of *changed* (denoted as c_i) and *unchanged* (denoted as $\neg c_i$). A class is changed, if either it did not exist in $v-1$ or its source/byte code is different in versions $v-1$ and v.

F **Nodes:**

For each class a_i, there is also a node of type F in the model. A F_i node, corresponding to the event of fault introduction to the class a_i, has two possible values of *faulty* (denoted as f_i) and *sound* (denoted as $\neg f_i$). A class is said to be defective (or faulty) if to fix a bug in the system one needs to change that class. From another perspective, a class is faulty if in exhaustively testing its input state, there exists at least one test case that fails.

T **Nodes:**

For any test case t_i in the the input test suite, a node of type T exists. A T_j node, corresponding to the event of executing the test case t_i, has two possible values of *failed* (denoted as t_i) and *passed* (denoted as $\neg t_i$).

4.3.2 Edges

The exact design of arcs in the BN model should be based on design decisions regarding the relations that need to be modelled. Each arc in a BN indicates a causal relation between variables of two connected nodes. The framework overview (presented in Section 3.1) outlines two different sets of edges that need to be included in the model, based on identified causal relations and conditional independencies. There are also flexibilities regarding the inclusion of some edges. In each realization, the exact set of included edges should be decided.

CF **Edges:**

For each class a_i, there is an edge CF_i from the node C_i to the node F_i. These edges indicate that a change in class a_i can potentially cause the introduction of faults to the same class.

Changing a class a_i is likely to impact other classes as well. Therefore, changing a_i can potentially cause another class a_k to become defective. Such a causal relation could be modelled as a C_iF_k edge from C_i to F_k (referred to as an impact edge). However, in this realization, this set of edges are decided to be excluded from the model. This design decision is justified mainly by considering complexity issues.

Including impact edges in the model adds to the complexity in two ways. First, the complexity of the probabilistic inference algorithms is affected. The running time of inference algorithms in a BN is heavily impacted by the topology of the network. Specially, the number of parents for each node has a significant impact on the running-time. Also, considering that the size of a CPT associated with each node grows exponentially with the number of parents, there are also memory issues with nodes with more than one parent. Second, should these edges be included, estimating the many numbers required in the huge CPTs of F variables would be a complex process. Having impact edges, the F nodes would be conditioned on potentially many variables and hence would have huge CPTs. Each cell in such CPTs indicates the probability of a class a_k becoming faulty given that a combination of the classes affecting it have changed. Although assumption of independence (*e.g.,* Noisy-OR assumption) can help reduce the table sizes, such assumptions here would be hard to justify. The impact of different classes on one class are not by no means independent from each other. Making estimations needed for change impact is a hard task that adds to the overall complexity of the implementation.

These additional complexities are justified only if inclusion of the impact edges is likely to lead to a significant improvement in the performance of the technique. If changing a_i introduces a bug that makes a_k faulty, chances are slim that a_i itself is not faulty. In case a_i is indeed faulty, the impact edge is helpful only if none of the test cases covering it reveal the fault but a test case covering the a_k does. Such a situation, although possible, is not prevalent. The additional complexity, hence, is not necessarily justified. Moreover, other less sophisticated mechanisms of considering change impact can be included in the model (as later descriptions reveal). Hence, the change impact edges are are decided to be excluded from the model. This decision is based on intuition (rather than empirical evidence) but such decisions are commonplace in software systems.

FT **Edges:**

For each test case t_j in the input test suite and for each class a_i covered by t_j, there exists an edge F_iT_j from the F_i node to the T_j node. These edges indicate that if a test case exercises a particular class, it can potentially reveal faults in that class, should there be any.

There is no edge from an F_i variable to a T_j if t_j does not exercises a_i. This is based on the assumption that a test case can not find whether a class is faulty unless it exercises it. This assumption follows directly from the given definition of a faulty class. A class is said to be faulty only when there could exist a test case covering that class that fails. This assumption seems reasonable from a practical point of view as well.

4.3.3 CPTs

Each node in the BN has a CPT associated with it. A CPT is a tabular representation of the conditional probability distributions for the random variables of a BN. For specifying a CPT, one needs to indicate the probability of each outcome of the variable, given each combination of the values of all its parent variables. Considering that all variables are binary in the proposed framework, a node with n parents has a CPT table of size $2 * 2^n$. Because in a binary variable $P(x) = 1 - P(\neg x)$, finding the values for 2^n of those cells suffices.

CPT of C Nodes:

A C_i node in this design is not conditioned on any other variables. Its CPT table, therefore, has only $2^0 = 1$ cell to fill. That cell is the $P(c_i)$. As described before, this value is used in this realization to indicate the probability of an *error-prone* change (rather than the plain probability of the change which would be simply zero or one). To estimate such probabilities, the amount of change can be heuristically used.

Given a metric $change(a_i)$ which for each class a_i gives a number between 0 and 1, indicating the percentage of the class modified from version $v-1$ to v, this CPT is specified by:

$$P(c_i) = change(a_i)$$

The metric function $change(a_i)$ is based on one of the potential measurement techniques described in Section 4.2.1. GNU diff algorithms can be used to find changed lines of code; the ratio of the changed lines of code to the total number of lines provides a number between zero and one that can be used here. `Sandmark`'s Constant Pool differencing algorithm, also, gives a normalized number for each class indicating its change; those measures can be directly utilized as the *change* metric.

CPT of F Nodes:

An F_i node in this design is conditioned on only one other variable, namely C_i. As such, there are $2^1 = 2$ values to estimate: $P(f_i|c_i)$ and $P(f_i|\neg c_i)$. These two values indicate the probability of the class a_i being faulty respectively when it is changed and when it is not. Naturally, the first number needs to be much higher than the second for all the classes. As such, the impact of the first number on the results is much higher than the second one. Both these probabilities are influenced by the quality of the class a_i. The fault-proneness of classes is estimated using metrics of software quality. As mentioned in Section 4.2.2, univariate models of fault-proneness (those relying only on one metric) are decent predictors of quality which are significantly less complex to create compared to multi-variate models. The univariate models are used in this implementation.

Given a metric of software quality $qm(a_i)$ which for each class a_i in the system gives a relative number indicating its quality from a certain perspective, this model estimates:

$$P(f_i|c_i) = \frac{\alpha\, qm(a_i)}{\max(qm(a_x))} + \delta_1,\ (\alpha + \delta_1 \leq 1)$$

where the function $\max(qm(a_x))$ denotes the maximum value of the metric qm across all the classes in the program. The α and δ_1 are two given constants the sum of which is less than or equal one so that the right hand side of the formula is in the shape of a probability number. The dominator in this formula is a simple normalization factor. Constants α and δ_1 are there to bound the probability of fault introduction. The constant δ_1 sets a minimum for the fault-introduction probability given that a class has changed, regardless of its quality. α sets an upper-bound for fault-introduction probability in the system of question.

Estimating the $P(f_i|\neg c_i)$ values is also required. The important invariant here is that these values should be much less than those of $P(f_i|c_i)$. A class can be faulty without being changed due to one of the lesser likely situations of i) being impacted by other changes, ii) change in its specification, or iii) containing residual faults. The third situation is not of interest in our problem. The residual faults would not be identified through re-execution of test cases that have been previously executed. The second item is hard to account for through measurement techniques; for most software projects such measurements are not available. The first situation, however, could be estimated more practically. A class is likely to be impacted by the change in other classes if it is highly coupled to others.

Given a metric of coupling $cm(a_i)$ which for each class a_i in the system gives a relative number indicating its degree of coupling with other classes, this model estimates:

$$P(f_i|\neg c_i) = \frac{\beta\, cm(a_i)}{\max(cm(a_x))} + \delta_2,\ (\beta + \delta_2 \ll \alpha + \delta_1 \leq 1)$$

where, the function $\max(cm(a_x))$ denotes the maximum value of the metric cm across all the classes in the program and β and δ_2 are two given constants. Again, β and δ_2 are there to bound the probability of fault introduction given a class has not changed. The constant δ_2 (which could be simply set to zero) sets a minimum for the fault-introduction probability given a class has changed; β sets an upper-bound for it.

The important invariant here is that $\beta + \delta_2 \ll \alpha + \delta_1$. This is what embeds in the model the fact that a changed class has more potential to contain faults than one without changes. Let $\gamma = \frac{\alpha+\delta 1}{\beta+\delta 2}$, such a parameter captures the extent to which the presence of change increases our belief in the presence of faults. By adjusting γ we can control the degree to which the presence of change in an element raises our belief in its fault-proneness.

CPT of T Nodes

A T_i node in the models of this framework can have more than one parent. If test case t_i is covering classes $a_1, \ldots, a_n$, the normal representation of the CPT has 2^n cells, each containing a $P(t_i|F_1, \ldots, F_n)$ value. As mentioned in Section 3.1.4, the Noisy-OR assumption is used to reduce the sizes of the CPT tables for T variables. The assumption is that the relation of a test case to a particular class is independent from its relation to any other classes. It can be argued that the ability of a test case to reveal faults in one class is not related to its fault revealing ability on the other classes, hence the assumption.

The Noisy-OR assumption reformulated to this problem indicates that

$$P(t_i|f_1, \ldots, f_n) = 1 - (1 - P_0) \prod_{1 \leq j \leq n} \frac{(1 - P(t_i|f_j))}{1 - P_0} \tag{4.1}$$

The $P_0 = P(t_i|\neg f_1, \ldots, \neg f_n)$. The P_0 (called the *leak* value in the terminology of Noisy-OR assumption) is the probability of a test case revealing faults even though all of its covered classes are sound. Theoretically, such an possibility can not be completely ruled out. The information of coverage might be incomplete, for instance, causing a test case to fail even though no coverage is known. For the consideration of such unlikely situations, a very small constant value can be assigned to this parameter (*leak*). The Formula 4.1 is a straight-forward application of the Noisy-OR assumption[3].

Formula 4.1 simplifies the CPTs of T variables. To build those CPTs, one only needs to estimate the individual $P(t_i|f_j)$ values in that formula. Given a function $cov(t_i, a_j)$ which for each test case t_i and class a_j gives the percentage of a_j covered by t_i, one can estimate:

$$P(t_i|f_j) = cov(t_i, a_j)$$

Most of the tools implementing Bayesian Networks allow specifying the CPT tables in the Noisy-OR form. That is, through specifying the effect of each parent on a node separately. Even so, they usually use Formula 4.1 to get the normal representation of the CPT before performing the inference. Therefore, the memory problem is not fully solved. There is only a limited number of parents a node can have. Having 20 parents for a node, for example, leads to 2^{20} numeric memory cells for each node which is on the edge with the current technology. A test case, however, can potentially cover much more classes. Modeling all the coverage relations imposes a great computational and space complexity. To cope with this problem, one can prune the parents of a node. If a test case covers more than a certain number of classes, those classes that have less coverage can be removed from the parents of the corresponding T variable. This strategy keeps the effects of pruning minimal because a test case with low coverage on a class is not that likely to find bugs in the first place.

4.3.4 Design Summary

Figure 4.1 depicts an overall design of the proposed BN model for this realization of the framework.

In sum, the design of the proposed BN is as follows. Each class a_i has a C_i and a F_i node, respectively representing the change and fault introduction in a_i. The F_i node is the child of C_i node and only that node. Then, for each test case t_j, there exists a T_j node which represents the potential results of its execution in

[3] interested reader is referred to any book on BNs [70, 101]

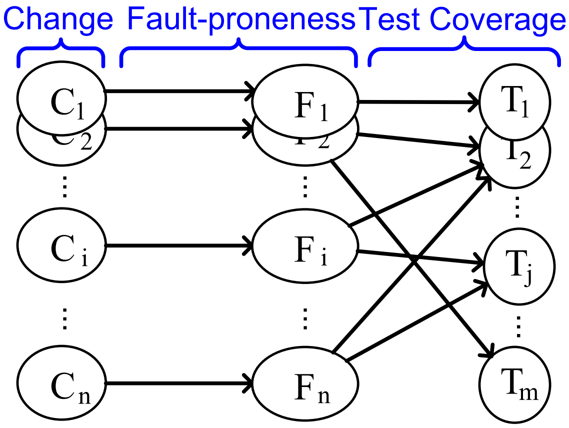


Figure 4.1: An Overview of the Proposed BN Design

regression testing phase. A T_j node is a child of a F_i variable only if class a_i is covered by t_j. To estimate the distribution of C nodes, change data is used; for F nodes, univariate models of fault-proneness are employed; and for T nodes, the coverage data in conjunction with the Noisy-OR assumption are utilized.

4.4 Tool Support Implementation

The last step in the realization process is to build a tool that implements the framework. Here, an implementation of the code-based realization for Java programs is briefly discussed. To implement the proposed code-based design, the approach presented in Section 3.2.3 is used as a basis. The following details the implementation specifics of each involved component, as shown in Figure 3.2.

4.4.1 Collecting Measures

Collecting the required metrics from software systems is a well-studied subject. The implementation of data gathering mechanisms, however, entail many technical details. Fortunately, many tools have been developed to perform different measurements on software systems. Many of these tools are open source software that can be used for research purposes.

For change analysis, two different tools have been used in this implementation: `UNIX diff` and `Sandmark` [114]. The first tool, easily accessible in all UNIX operating systems, computes the textual differences between two files. For a Java program, each file represents a class. The amount of change in a class, therefore, is easily measured by feeding the two consecutive versions of the class to the `UNIX diff` application. The results are analyzed to find the number of lines changed; a number which is then divided by the total number of lines to get a single value between 0 and 1. `Sandmark`, mainly developed as a watermarking tool, implements more sophisticated algorithms of code differencing. This tool which works on the byte code of Java files, compares *.class* files against each other and gives a score of similarity between 0 and 1. The resulting numbers are subtracted from 1 to get the difference score. That score is used as an indication of the amount of change.

To gather quality metrics, a tool called `ckjm` [24] is used. This tool computes the metrics from the Chidamber-Kemerer (CK) metrics suite [21]. The CK suite, as explained in Section 2.2.2 is the most widely-used set of software quality metrics for OO systems. `ckjm` provides an easy to use command-line that gets a Java byte code (.class files) and outputs a report listing the measures for different classes.

For gathering coverage data, a tool called `Emma` [46] is employed. This tool instruments the code, either off-line or on-the-fly, to gather coverage of test cases. It provides XML reports that can be configured to show coverage data in different levels of granularity, including that of a class. Such XML reports, are then parsed to get the required coverage data. In addition to `Emma`, the implemented tool can also read the coverage reports from an instrumentation tool called `Sofya` [75]. This tool instruments a Java program such that during its execution, the exercised basic-blocks are reported in trace files. These trace files are then read to compute the coverage in basic-block level. Those data are finally abstracted to the class level to determine the percentage of blocks covered in a particular class.

This implementation relies on a report-based architecture. Each of the aforementioned tools are first executed separately to generate reports that contain the results in different formats. The reports are then read by programs developed to understand an specific report format. Those programs create abstract data structures representing the gathered data in logical levels. Reports from different tools can be used to build these common data structure. The data structures are used later on to build the BN. This enables us to get the required data from different tools, as long as they can be presented in the same data structures.

4.4.2 Building the BN

To work with Bayesian Networks, a tool called `Genie/Smile` [52] is used in this implementation. This tool has two main parts. `Smile` is a library that provides an API that can be called from other programs to use the tool. This API has many useful features: it provides support for C++, Java, and C#; it implements a handful of inference algorithms; it supports Noisy-OR nodes; and it exports and imports BNs to XML files. `Genie` provides a GUI that can be used to manipulate BNs visually. This tool can help testers browse the built BNs to understand the interworking of the approach. The GUI also helps debugging the implementation.

Having the detailed design on hand, and using the `Smile` library, building the BN is a straightforward task. First, the structure of the network (a graph) is built based on the gathered data and then based on the collected measures from the previous step, the CPTs are filled. The resulting BN is kept in a data structure in memory and also persisted in an XML file. The network is used by that later steps to perform the probabilistic inference.

4.4.3 Optimizing the Test Suite

Optimizing the test suite and outputting the final order is the last component of the system. For *probabilistic inference* component, the `Genie` tool is used. `Genie` implements many inference algorithms, from both exact and sampling algorithms. The implementation of inference algorithms in `Genie` is in C++ which makes it relatively efficient. The choice between the algorithms depends on the structure of the BN which in turn depends on the program under analysis. Simple experiments could be performed for a particular object to choose the inference algorithm. The next chapter of this thesis investigates that question empirically.

The results of the inference, provided as data structures in memory by `Genie`, are then used by developed tools to select and output a subset of test cases in iterations. The algorithm of this step is described in Section 3.2.3. In each iteration *stp* number of test cases are outputted based on their fault-revealing probabilities; then, the model is updated and the next iteration starts. Iterations halt when all needed test cases are outputted.

4.5 Summary

This chapter has described in detail how the general proposed BN-based framework of Chapter 3 can be realized for a typical system for which code is available. This realization is build around the software code. All required measurements are computed from the source/byte code. This realization is applicable to any system for which code is available and hence is applicable in most cases. This particular realization is of interest also because it is comparable to the other approaches to regression test optimization problem from the literature which are also code-based.

The next four chapters in this dissertation empirically evaluate the presented code-based realization against other existing techniques. They also investigates the effects of different parameters present in the design and implementation of the code-based realization.

Chapter 5

Experimental Setup

The BN-based framework, as introduced in Chapter 3, aims to increase precision and provide flexibility. While the two previous chapters elaborated on the flexibility features built into the framework, enhanced precision can be demonstrated only through empirical experimentation. To evaluate whether incorporating many sources of information into one BN model improves on existing techniques, this thesis uses empirical studies. This chapter presents the setup for such experimentation.

The proposed framework is a basis to develop specific framework realizations, suitable for different environments. Chapter 4 provided one such realization for a typical system where code is available and measurements of the code can be taken. Many other techniques from the literature also focus on code-based systems. This facilitates the comparison of the approaches on the similar settings. For these reasons, the code-based realization is the subject of the experiments presented in this thesis.

Specific realizations can also address several instances of the general regression test optimization problem. Different instances of the problem can pursue different goals such as better fault detection, and better coverage. Of these, enhanced fault detection is the focus of most existing techniques. Instances of the general problem such as size-constrained test case selection and test case prioritization have much in common; knowing the probability of test cases finding bugs can lead to solutions to many instances. The approaches from the literature, with a few exception such as [73], mostly focus on one instance of the problem. Comparing against those techniques, hence, requires focusing on one instance of the problem.

The target realization can be best evaluated as a test case prioritization technique. There are some advantages in doing so. In test case prioritization, the resulting test suite contains all the test cases and hence all the estimated probabilities are taken into account. Evaluating the proposed approach as a selection technique requires thresholds, the choice of which would impact the final results. Using any particular threshold, some probabilities are not involved in the evaluation at all, rendering the results less expressive. Also, many of the estimations made in the realization described in the previous chapter are relative numbers, rather than absolute figures. The results from such estimations are more suitable for prioritization which concerns the relative positions of test cases. In view of these, code-based realization is evaluated as a test case prioritization technique.

This chapter discusses the experimental setup of evaluating the code-based realization as a prioritization technique. It starts with laying out the evaluation procedure in detail. Then target objects, used faults, involved techniques, and evaluation criteria are discussed each separately in a different section. The experimentatal setup introduced here is used in the three separate experiments that follow in the next three chapters. Those three experiments seek answering different questions; *Experiment 1* investigates the effects of different parameters present in the technique, *Experiment 2* and *Experiment 3* compare the proposed technique against others from the literature, respectively on mutation and hand-seeded faults.

5.1 Evaluation Procedure

The evaluation procedures undertaken in all the three experiments of the subsequent chapters are similar. The procedure enables performing controlled experiments. In each experiment, the performance from the target technique is compared against other techniques (as control samples). Independent variables of each experiments are identified and the corresponding results for dependent variables are observed. Statistical tools are used to evaluate the results. The common evaluation procedure is as follows:

(i) The target programs should be selected. For each program, a series of consecutive versions should be acquired. For each version, in addition to its source and byte code, its test suites are required.

(ii) The test suite of version v should be maintained to get a potential regression test suite for version $v + 1$. Maintaining, here, refers to pre-optimization regression tasks such as obsolete test case elimination, as described in Section 1.1.

(iii) For evaluating the fault-detection power of techniques, faulty variations of the programs are required. Either faults are naturally present in the target programs, or they should be inserted artificially. Automatic code mutation and manual fault-seeding could be used for fault insertion.

(iv) On the faulty programs and for each version, the complete potential regression test suites should be executed. Based on the test results, fault-matrices should be generated. These matrices detail for each faulty variation, what test cases have revealed the fault.

(v) The prioritization techniques involved in the experiment should be run. Each technique creates a different optimized regression test suite.

(vi) Based on a chosen criteria of evaluation and having fault-matrices, the resulting test suite of each technique should be evaluated. Statistical analysis can be also performed to further understand the implications of the obtained numerical results.

Each step of the procedure involves the use of some set of data to produce other data. The first two steps concern the required programs and their test suites, referred to as the *target objects*. The third step deals with *faults* necessary for the evaluation. In the fifth step, *involved techniques* are the focus. The last step involves the use of *Evaluation Criteria* for getting numerical results. The specifics regarding each of these notions is further elaborated in the following sections.

5.2 Target Objects

The target objects of the experiments need to be Java programs with at least a few versions available. For each version, source code and the complied byte code is required. Each version needs to have its test suite. The test suite of each version also needs to be maintained for the next version to get a potential regression test suite. Gathering all these data is a cumbersome time-consuming task. Researchers in this area have thus sought to ease the process by building ready-to-use repositories of experiment objects.

One such set of available data is Software-artifact Infrastructure Repository (SIR) [36]. SIR is, as its website describes[1], a repository of software-related artifacts built to support controlled experimentation with software testing techniques. This repository has been extensively used by other researchers for controlled experiments on test case prioritization techniques (*e.g.,* [38, 87, 111]). Using SIR, the time-consuming task of object preparation can be avoided. The SIR repository is used for the experimentation in this thesis. This choice facilitates the comparison of the obtained results with other techniques from the literature.

Five open source Java programs are provided in the SIR repository: *ant*, *xml-security*, *jmeter*, *nanoxml*, and *galileo*. The *ant* program is a Java-based build tool, extensively used in industry for building and deploying Java (especially web-based) applications. *Jmeter* is a Java application for load-testing and performance measurement. *Xml-security* implements some XML security standards. *Nanoxml* is a small parser for XML files in Java language. *Galileo* is a byte code analyzer application which can instrument Java code and gather trace files.

Table 5.1 lists for each target object the following attributes: the number of available consecutive versions of the program (Versions), the number of class files in the most recent version of the program (Classes), the total lines of code in the most recent version of the program in thousands lines (LOC (K)), the number of

[1] http://sir.unl.edu/portal/index.html

test cases available for the most recent version of the program (Tests), the total numbers of mutation faults considered for our object programs, summed across all versions of each program (Mutants), and finally the type of the test suite (Type).

Table 5.1: Target Objects and Their Attributes.

Objects	**Versions**	**Classes**	**LOC (K)**	**Tests**	**Mutants**	**Type**
ant	8	627	80.4	877	412	JUnit
jmeter	6	389	43.4	78	386	JUnit
Xml-security	4	143	16.3	83	246	JUnit
nanoxml	6	26	7.6	216	204	TSL
galileo	16	87	15.2	912	2494	TSL

The first three objects have JUnit test suites, and the last two have TSL (Test Specification Language) suites [100]. JUnit is a unit testing framework for the Java programming language. A JUnit test suite is itself a Java program. A typical test in JUnit is a function that first sets up a class to a particular state and then invokes one of its operations and asserts that the results are what is expected. JUnit test developers are encouraged to target one and only one functionality of the system in each test case. TSL is based on the category-partition method introduced in [100]. A TSL test case consists of an operation, its parameter values, and the environment variables. A TSL test script first runs the operations required for creating the environmental conditions; then, test case operations themselves are run; next, the results are validated; and finally, a clean-up is performed. Having test specifications written in TSL, tools exist to automatically generate test scripts.

This set of objects provides variety in many aspects. This set spans from small software such as *nano* to larger systems such as *ant* and *galileo*. The number of available test cases ranges from tens to hundreds, providing diversity in the test suite size. Most importantly, objects with two different test suite types are present in the repository.

5.3 Faults

To evaluate the performance of the proposed techniques for better fault detection, we require object programs containing faults. Note that only faults that can be found by available test cases are of interest. If faults can not be revealed by any of the test cases, they does not help our evaluation process. The available versions of the object programs, as released by their developers, do not have faults that could be found by their test suites. To get faulty variations of the objects, therefore, one needs to artificially insert faults into the programs.

The process of introducing faults to the programs can be performed in different ways. The objects in SIR come with two different types of faults: *mutation* and *hand-seeded* faults.

Mutation Faults:

Mutation faults are syntactic changes made to program source code to cause the behavior of the program to change. This notion has been introduced originally for mutation testing techniques of evaluating adequacy [17, 57]. These techniques insert simple syntactic code changes into the program, and then check whether the test suite can detect these changes (kill mutants). Any mutant that can be killed is a fault. Code mutation is inherently compiler-specific. Researchers have introduced mutation testing for different languages, including Java (*e.g.,* [5]). The representativeness of mutation faults have also been investigated (*e.g.,* [1]), revealing that mutation faults can in fact be representative of real faults.

In [35], Do *et al.* investigated the performance of prioritization techniques, measured based on mutation faults. Their study is also on the SIR Java objects. They have inserted mutation faults to those five Java programs and found mutants that can be killed. As the focus of their work also is regression testing, all the mutations in each specific version fall within the neighborhood of modified code. For the evaluations presented in the next three chapters, the same pool of mutation faults have been obtained from the authors of [35] and used. Using the same set of faults makes the results presented in this work further comparable to those of the mentioned study. The number of mutation faults available for each of our object programs is shown in Table 5.1.

Hand-seeded Faults:

The SIR repository has hand-seeded faults. A hand-seeded fault, as the name suggests, is one which has been manually inserted in the code. In [36], the authors describe their process of adding hand-seeded faults. They mention that to increase the potential validity of obtained faults, they insert faults by following fault localization guidelines (available in the SIR infrastructure documentation).

The number of available faults for most of the objects from SIR is not sufficient to enable any real experimentation (one or two in most cases). Only *ant* has several hand-seeded faults in different versions. When dealing with hand-seeded faults, therefore, the experiment investigates only *ant*.

Both families of mutation and hand-seeded faults have been used for the evaluation purposes. One of our experiments uses the automatically generated mutation faults while the other uses the hand-seeded faults.

5.4 Investigated Techniques

In the experimentation to follow, the code-based realization of the framework is compared against test case prioritization techniques from the literature. The following is a description of all investigated techniques and their implementation specifics.

Control Techniques:

The control techniques in a controlled experiment are those that represent the common practice or trivial solutions to the problem. There is one control technique utilized in the experiments: *original* ordering. Original ordering represents the common practice of running test cases in the original order that they appear in. Based on the technology used to develop a test suite, the test cases come with an original ordering which is used here as a control technique. The technique is referred to as *orig* in the diagrams.

Coverage-based Conventional Techniques:

Among the existing test case prioritization approaches, reviewed in Section 2.1.2, the conventional coverage-based techniques (*e.g.,* [111, 122]) are predominant. This family of techniques represents the earliest line of research in test case prioritization. The performance of these techniques has been extensively studied empirically (*e.g.,* [39, 43, 111, 122]); also, most other proposed approaches compare their results against these techniques as a benchmark. There exist a variety of coverage-based prioritization techniques which use different metrics of coverage and algorithms of ordering. This experimentation includes two common coverage-based techniques.

One factor that differentiates coverage-based techniques is the level of granularity on which coverage data is gathered. Empirical studies, however, show that the effect of this factor is not significant in many cases (*e.g.,* [43]). Therefore, the performed experiments focus on one level of granularity, that of methods. Note that method level data are more fine-grained that the information used in code-based BN realization. Nevertheless, method-level data are employed here because using class level coverage data for prioritization has not been reported in the literature.

The use of feedback mechanisms is another factor differentiating prioritization techniques. This factor, contrary to the level of granularity, is found to significantly affect the results in many cases (*e.g.,* [39, 43]). Therefore, in the evaluation, two versions of coverage-based techniques are included: one without the feedback mechanism and one with feedback. The first variation is called *Method Coverage* (MC) technique and the latter is called *Method Coverage Additional* (MCA) technique.

BN Techniques:

The BN techniques are, here, the subject of the controlled experiments. Different variations of the BN technique are used in the three experiments that follow in later sections. These variations mostly differ in the value of the parameters that are present in the implementation presented in Chapter 4. Much like the conventional coverage-based techniques, investigated variations of the BN-based framework can be categorized into two techniques: one with feedback and one without feedback.

The feedback mechanism is part of the BN-based framework (as described previously in Section 3.2.3) which can be controlled through the *stp* parameter. Setting this parameter to one causes the algorithm to add test cases one by one and evaluate the effect each time. This corresponds to the feedback mechanism. Setting the parameter to the size of the test suite creates a non-feedback implementation. Any other number would result in partial feedback. The performed experiments include a feedback and a non-feedback implementation, respectively called BN and BN Additional (BNA), are always present. For BNA, different experiments use different values of *stp* parameter; the exact values are mentioned in each experiment.

Table 5.2: Techniques Involved in the Experiments.

Label	**Description**	**Feedback**	**Coverage**	**Family**
orig	the original ordering of test cases	-	-	Control
MC	ordering based on Method-level Coverage	No	Method	Coverage-based
MCA	ordering based on Method-level Coverage Additional	Yes	Method	Coverage-based
BN	code-based realization of the BN-based framework	No	Class	BN-based
BNA	code-based realization of the BN-based framework with feedback	Yes	Class	BN-based

Table 5.2 summarizes the involved techniques. The first column is the label by which the techniques is referred to in the rest of the chapter. The second column gives a brief description of the techniques. The third column indicates whether feedback mechanism is used in the technique and the fourth column shows the granularity level of the coverage data. Finally, the last column specifies the family to which a technique belongs. The first experiment involves merely the BN-based family of techniques but the two later experiments involve all the three families.

5.5 Evaluation Criteria

Evaluating the performance of different techniques requires criteria on which to compare the techniques. The performance of prioritization techniques (or regression techniques in general) has many different facets. First and foremost, the goal of prioritization is to enhance the rate of fault detection and hence an important evaluation criteria concerns the fault detection speed. Moreover, like any other computer algorithm, the running-time required by the technique is of interest. In addition, other factors such as the required manual effort to set up the technique could be all considered. Many of these criteria conflict with each other, so that enhancing one leads to the deterioration of the others. One way to account for these different criteria is by using cost-benefit models that translate the benefits and costs of each techniques to quantifiable metrics, *e.g.,* time or money.

There are comprehensive models of cost-benefit analysis in the literature of test case prioritization (*e.g.,* [38, 107]). Nevertheless, they are not used in this research work for the following reasons[2]. First, the resulting numbers from those models are hard to analyze. Because those models take many factors into consideration, one can not easily understand why a technique is performing well or poorly. Those models, although helpful for industrial comparison, are not descriptive enough for the goals of experiments performed in this research work. Moreover, to be able to use such models, many figures related to the processes of the target software project are required. For the target objects of this experiment, and for many other software projects, such figures are not available. To use those models, therefore, one needs to rely on guesstimates that are not necessarily accurate. Bearing in mind the fact that such figures have huge impact on the models, the results should always be interpreted with great care.

Instead of using cost-benefit models, one can use evaluation metrics that have been widely used in prioritization literature. Two evaluation metrics are used here: APFD (Average Percentage Faults Detected) [111] and running time of the algorithms. While the APFD metric provides insight to the benefit of each technique, the running time helps compare the costs.

APFD: This metric calculates fault detection rate by measuring the weighted average of the percentage of faults detected over the test suite execution period. APFD values range from 0 to 100 where higher values indicate faster fault detection rates. Consider that we plot the the percentage of faults detected versus the percentage of the test suite executed (using the data from fault-matrices). The area under the resulting curve provides a measure of how fast bugs have been found. Informally speaking, APFD metric is the area below such a curve. More formally, let $T =< T_1 \dots T_n >$ be a test suite with n test cases, and let $F_1 \dots F_m$ be m faults revealed by those test cases. Let TF_i be the index of the first test case in T that reveals F_i. The APFD for test suite T is given by:

$$APFD(T) = 1 - \frac{\sum_1^m TF_i}{nm} + \frac{1}{2n}$$

The precise definition of APFD and its justification can be found in [111].

Running-time: The running time of each implementation is computed on a Windows system with a Pentium(R) 4 CPU and 2GB of RAM.

Finally, note that different parts of this chapter pursue different goals in experimentations. Not all of the above evaluation criteria are of interest in all the cases. Each experiment reports the metrics of interest in its particular case. These metrics of evaluation are in fact dependent variables of the controlled experiments.

[2] Note that such models are indeed used for evaluation of this framework in published papers [37]. This thesis, however, does not report those data.

5.6 Summary

This chapter has presented the experimental setup for three sets of experiments that will follow in the next three chapters. Those experiments are designed to evaluate the performance of the code-based realizations of the BN-based framework as a prioritization technique. The chapter has started with the evaluation procedure. It then discussed the involved techniques. Method level coverage-based techniques (an existing approach from the literature) are included as a baseline in addition to two variations of the BN-based framework for which the implementation is detailed. Mutation and hand-seed faults have been also introduced as alternative ways of artificially adding faults to systems. Finally, the chapter introduced APFD and running time as the metrics of comparison.

The next three chapters investigate three sets of research questions through empirical experiments. All those experiments are conducted based on the experimental setup outlined in this chapter.

Chapter 6

Experiment 1: The Effect of the Parameters

The code-based realization, as presented in Chapter 4, has a few parameters that could be utilized to adapt the behavior of the approach to the specifics of a special environment. The effects of these parameters could best be understood through empirical studies. A controlled experiment is performed to evaluate such effects; the results from that experiment are reported in this chapter. The goal of the controlled experiment is to compare different implementations of the proposed framework and to understand the effects of its underlying parameters. The followings are research questions pursued through this experiment:

- **RQ1:** What is the effect of the feedback mechanism on the performance of the framework?
- **RQ2:** What is the effect of using different measurement techniques (*i.e.,* change analysis and fault-proneness models) on the performance?
- **RQ3:** What is the effect of adjusting parameters of the framework realization (*i.e.,* inference algorithm and *stp* parameter)?

This chapter reports on a controlled experiment designed to answer these questions. The rest of this chapter first describes the experiment design, including variables and measures and then explicates threats to the validity. Obtained results follow next and at the end, a discussion on the obtained results is presented.

6.1 Experiment Design

In a control experiment, one first needs to identify the independent variables that affect the system. The effects of the independent variables are then evaluated based on the dependent variables. The following first details the involved variables and measures, then explains the particular staged design of this experiment.

6.1.1 Variables and Measures

Independent Variables

To investigate the aforementioned research questions, the following independent variables are needed.

V1: Feedback Mechanism- The decision on whether to use feedback mechanism produces two variations of the techniques: BN referring to the case where feedback is not used, and BNA, the *additional* version, where feedback is included[1].

V2: Change Analysis Technique- Two alternative techniques for gathering change information is evaluated here: *Unix diff* and *Constant Pool diff*. These two techniques have been discussed in Section 4.2.1. The V2 variable captures these two techniques with the possible values of D (DIFF) and CP (Constant Pool).

V3: Fault-proneness Model- The fault-proneness models from literature (*e.g.,* those surveyed in [10, 55]) estimate fault-proneness using metrics of quality (one metric in the case of univariate models). This experiment evaluates five common metrics from Chidamber and Kemerer suite [21]: CBO (Coupling between Objects), DIT (Depth in Tree), LCOM (Lack of Cohesion), RFC (Response for Class), and WMC (Weighted Method Complexity). Therefore, V3 variable, characterized by the selected metric suite, can have five different values.

V4: Inference Algorithm- BN Inference algorithms fall into two categories: *exact* (EXC) and *sampling* (SMP) algorithms. One sampling algorithm (Epis [125]) and one exact algorithm (Lauritzen [81]) are included in this experiment.

V5: *stp* Parameter- The *stp* parameter is also a subject of this experiment. The following five values are chosen as the domain of this variable: 1, 5, 10, *adjustive* 10, and *adjustive* 20. The last two values are *adjustive*, meaning that the parameter is set to 1 initially and increased gradually (and uniformly) as the test cases are added to the ordered list. The value reaches the specified number at the end of the ordering process. For example, if we have 500 test cases with *adjustive 10* value, *stp* is set to 1 initially. After 500/10 = 50 iterations (when 50 test cases are added), *stp* is increased to 2. Then, after 25 more iterations (when another 50 test cases are added), the parameter is increased to 3 and so on until the last 50 test cases where *stp* is set to 10. Using this schema, the earlier test cases in the ordered list are chosen using small *stp*s and hence more carefully. As we go down the order list, the importance of the test cases is reduced and hence larger *stp*s are chosen.

Dependent Variables

The two evaluation metrics of APFD and running-time, as described in Section 5.5, are chosen as the dependent variables of this study.

[1] Note that the use of feedback can be controlled through *stp* parameter.

6.1.2 Staged Design

In performing an experiment with five different variables, one problem is the interaction between the variables. One can investigate all the possible combinations of the variable values. This makes the space of the experiment large and hard to analyze. Another path, taken in this experiment, is to divide the experiment into different stages that each target different variables. In each stage, variables that are less likely to interact with the target variable should be fixed to a value of choice.

This controlled experiment has been performed in four different stages, each of which targets one of the variables V2 to V5. We expect that the feedback mechanism (V1) has a major impact, and hence, it has been treated differently. Both of V1 values (BN and BNA) are included in the first two stages. Besides V1, in each stage the value of all the variables except the one targeted in that specific stage are fixed. For all the values of the variable under experiment, the dependent variables are measured and compared statistically. In some stages, *running-time* dependent variable is almost identical among all alternatives and is not reported.

Table 6.1 lists for each variable, the fixed values in all the stages. The stage which concerns a specific variable is marked as "Target". The choice of the actual value for a fixed variable in each stage is based on either our expectation of the optimal value, or the results from a previous stage. For example, CBO has been initially chosen for V3 because it is widely reported to be a better indicator of fault-proneness (*e.g.,* [55]). However, after finding the optimal values in Stage 2 for each specific object, the value of the variable has been changed for Stages 3 and 4.

Table 6.1: Assigned Values to Each Variable for the Four Stages.

Variable	**Stage 1**	**Stage 2**	**Stage 3**	**Stage 4**
V1	Target	Target	BNA	BNA
V2	Target	Constant Pool		
V3	CBO	Target	*ant, galileo*: DIT *xml-security*: RFC *jmeter, nanoxml*: WMC	
V4	*ant*: Sampling the rest: Exact		Target	*ant*: Sampling the rest: Exact
V5	1			Target

The first research question (RQ1) is pursued through Stages 1 and 2 of the experiment where the feedback variable is included. The second research question is divided into two parts of change analysis and fault-proneness model, addressed in Stages 1 and 2, respectively. The third research question is also divided into two parts of inference algorithm and *stp* parameter, addressed in Stages 3 and 4, respectively. These two parameters are of interest only for BNA; *stp* is meaningful only for BNA and the inference algorithm mostly affects running-time which is not an important issue for BN. Therefore, V1 is set to BNA for the last two stages.

6.1.3 Threats to Validity

This section describes the construct, internal, and external threats to the validity of this study as well as the cautions taken to limit the effects of these threats.

Construct Validity. The dependent measures used in this experiment are not complete in terms of capturing all the costs and benefits associated with the different alternatives. Performing accurate cost-benefit analysis is dependent on the specific context where a given technique is used. However, both cost and benefit are to some degree captured in the two dependent variables described earlier.

Internal Validity. There could be several other factors that influence the results and affect the inferences made from the results. The tools developed/used to collect data can potentially have defects. By validating the tools on several simple Java programs, this threat is restricted. More importantly, the staged procedure of the experiment can influence the result. Changing the chosen fix values can (although unlikely) affect the results for the other variables. To restrict this problem, the variables are chosen such that they have little logical dependency on each other. Also, small experiments have been conducted to validate the effect of variables on each other; the experiments indicate relative independence of the variables.

External Validity. There are also threats to the external validity of the experiment. The programs studied here are not large by today's industry standards and they are all open source software. Complex industrial programs with different characteristics may be subject to different concerns. Also, the testing process used for the open source software is not representative of all possible processes used in industry. The other threat to generalizing the findings is the use of mutation faults for evaluation. Although using real faults is preferable, they are typically not available in large numbers; thus, researchers use faults created by mutation tools instead. It has been shown that mutation faults can be representative of real faults [35], although the result from such studies should be interpreted with care. Finally, the tools used in this study are prototypes and thus may not reflect tools used in a typical industrial environment.

6.2 Obtained Results

This section presents the obtained results from the outlined experiment along with the statistical analysis of the data. Figure 6.1 plots an overview of the results for all the five stages. Each column represents one stage and includes 5 boxplot diagrams, each corresponding to one experiment object. In each diagram, the independent variables corresponding to that stage are enumerated in the horizontal axis. The vertical axis shows the APDF metric. For each version of an object, 15 sets of mutants are randomly chosen each having a random size of 1 to 10. The boxplots illustrate the distribution of these 15 sets for all the versions of each object. The reminder of this section looks into the specifics of the results for each stage.

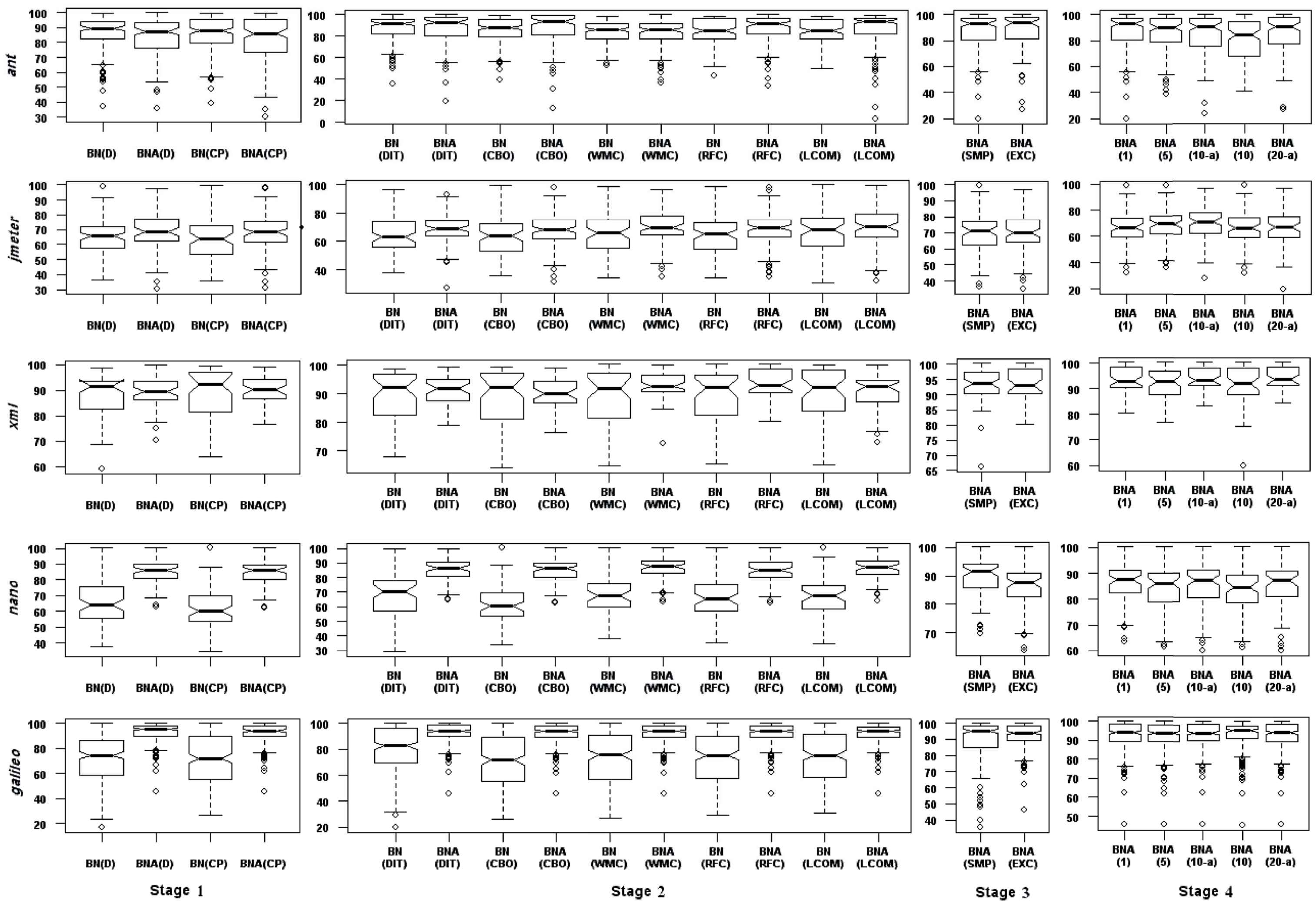


Figure 6.1: Boxplot Diagrams of the APFD metric for All the 4 Stages

6.2.1 Stage 1

In this stage, the effects of using different change analysis techniques (V2) and also feedback mechanism (V1) are examined. The first column of the Figure 6.1 shows the results related to this stage. The letters in bracket indicate the values of V2 where D stands for Unix diff technique and CP refers to the Constant Pool.

Looking at the boxplots, the change analysis technique (V2) does not seem to affect the results significantly. The use of feedback mechanism (V1), however, seem to have an important effect in some cases. To validate these informal observations, Kruskal-Wallis statistical test [106] is utilized here. Kruskal-Wallis is an statistical tool for one-way analysis of variance by ranks. It is a non-parametric method for testing equality of medians in different data sets. Intuitively, it is identical to a one-way analysis of variance (ANOVA) with the data replaced by their ranks. This test is a non-parametric method and hence does not assume any distribution for the population, unlike ANOVA. This test is used here because the assumptions of ANOVA does not hold for the data in this experiment; the data has too much variance to assume a normal distribution.

Table 6.2 illustrates the results of the statistical test in two parts, investigating the effect of V1 and V2. In the first part, the realizations of the technique with different values of V1 are compared against each other. In the second part, those with different V2 are examined.

Table 6.2: Kruskal-Wallis Test Results for Stage 1.

V1				
Program	**BN(D) vs BNA(D)**		**BN(CP) vs BNA(CP)**	
	chi.	p-val.	chi.	p-val.
ant	0.9	0.323	0.08	0.762
jmeter	2.5	0.114	3.8	0.049
xml-security	0.03	0.8	0.14	0.701
nanoxml	56	< 0.0001	66	< 0.0001
galileo	130	< 0.0001	132	< 0.0001

V2				
Program	**BN(D) vs BN(CP)**		**BNA(D) vs BNA(CP)**	
	chi.	p-val.	chi.	p-val.
ant	0.04	0.79	1.1	0.287
jmeter	0.7	0.395	0.05	0.827
xml-security	0.82	0.363	0.17	0.680
nanoxml	3.3	0.069	0.01	0.942
galileo	1.34	0.247	0.20	0.653

The results of the statistical test reveal that not a single case of significant difference for V2. Clearly, this variable is not an important factor, meaning that different change analysis techniques do not significantly differ from each other. Evaluating V1, on the other hand, reveals significant differences. BNA results in significantly better APFD measures for *nanoxml* and *galileo*, the two objects with the TSL test suite. However, for other objects (the ones with Junit test cases), the differences are not statistically significant. This implies that use of the feedback mechanism does indeed improve the performance but not in all cases.

Table 6.3: Average Running-time (in sec), Stage 1.

Program	**BN(D)**	**BNA(D)**	**BN(CP)**	**BNA(CP)**
ant	5	462	5	497
jmeter	6	71	4	66
xml-security	0.7	2.2	0.7	2.2
nanoxml	2	6	2	6
galileo	6	71	5	66

Table 6.3 reports the running-time of the four techniques. It is evident (as expected) that V2 does not have meaningful effect on this dependent variable. V1, on the other hand, has a significant impact on the running-time. The running-time of BNA technique is between 3 to 100 times larger than that of BN. Using feedback can assist early fault detection in some cases, but it does require longer running-time.

6.2.2 Stage 2

The main concern in Stage 2 is the fault-proneness metrics (V3). The second column of Figure 6.1 illustrates the results related to this stage for both BN and BNA techniques. The boxplot diagrams suggest that the V3 variable only occasionally has a significant impact. A five way Kruskal-Wallis statistical test (dt=4) is used to find out if the differences between the techniques are statistically significant.

Table 6.4: Kruskal-Wallis Test Results for Stage 2.

Program	**BN**		**BNA**	
	chi.	**p-val.**	**chi.**	**p-val.**
ant	7.6	0.109	30.44	< 0.0001
jmeter	4.3	0.36	2.4	0.65
xml-security	0.06	0.963	13	0.011
nanoxml	12.4	0.015	2.6	0.623
galileo	30.7	< 0.0001	1.3	0.861

The results are shown in Table 6.4. Unlike Table 6.2, here V1 variable is not separately investigated because the same pattern as that of Stage 1 is observed.

To control merely for V3, the test is performed among the five BN and BNA alternatives separately. That is the variations of the BN technique are compared together, BNAs to each other. The results indicate that V3 dose not change the performance significantly in most of the cases but occasionally does.

There are two cases where significant differences are observed: BNA technique for *ant* and BN for *galileo*. The reported statistical test is among five populations. The interpretation of such a test is that there is at least one technique which is statistically different from at least one other technique. To gain a better understanding of the results, we performed Bonferroni adjustment [106] for the two cases of BNA in *ant* and BN in *galileo*. This test adjusts the results of the statistical test to compare the results pair-wise. According to this test and in the significance level of 1%, for *ant*, using WMC is significantly better than all the others but the rest of the metrics are not different from each other. For *galileo*, DIT is significantly better than all the other alternatives but the rest are statistically similar.

6.2.3 Stage 3

In this stage, the inference algorithm (V4) is concerned. The third column in Figure 6.1 shows the APFD metric for this stage. The technique labelled as EXC uses the exact algorithm and the one labelled SMP uses a sampling algorithm. In principal, the APFD metric for different algorithms should not be much different. The boxplot diagram suggests that the actual results support this point except for *nanoxml* object. Performing Kruskal-Wallis test for this specific object, we observed that the $p-value$ is 0.007 which means the result is significant in 1% level but not in 0.1% level. Interestingly, here, the sampling algorithm has a better APFD value; an observation for which no solid justification is available.

Table 6.5: Average Running-time (in sec), Stage 3.

Program	***ant***	***jmeter***	***xml-security***	***nanoxml***	***galileo***
BNA(EXC)	871	2	1	8	66
BNA(SMP)	496	11	12	105	1434

The choice of inference algorithm influences the running-time to a large extent. Table 6.5 lists the average running-time of the two BNA techniques. There is a mixed pattern observed in the results. While the exact algorithm (EXC) performs quite slower for *ant*, it is much faster for the other objects. This result, although surprising at the first glance, is easily justifiable. As the size of the network grows, the sampling algorithm's complexity does not increase dramatically whereas in the exact algorithm, it can grow exponentially. Therefore, for small systems, the sampling algorithm is relatively slow. That is because the time necessary for generating samples, etc. is more than traversing the network. For large systems, on the other hand, it performs faster because the complexity does not grow exponentially. Because *ant* is a notably larger system than the other objects, SMP performs faster on it. It is noteworthy that the specific exact algorithm used in this experiment traverses the network in a forward search fashion, and hence, the number of nodes in the first layer (*i.e.,* number of classes) have a larger impact on the running-time than the nodes in the last layer (*i.e.,* number of test cases).

6.2.4 Stage 4

In this stage, the *stp* parameter is the subject of the analysis. The last column of Figure 6.1 depicts the boxplot diagrams related to this stage. The effect of *stp* parameter is expected to be two-fold. It can potentially affect APFD values and it certainly changes the running-time.

Table 6.6: Kruskal-Wallis Test Results for Stage 4.

	ant	***jmeter***	***xml-security***	***nanoxml***	***galileo***
chi.	21.6	3.3	4.1	4.2	1.3
p-val.	0.0002	0.5	0.395	0.372	0.866

Table 6.6 shows the result of the statistical test on APFD for all the five objects. The differences in APFD are not statistically significant except for *ant* for which the value of 10 for *stp* results in significantly lower APFD. It could be concluded that the specified values for *stp* parameter do not have a significant impact on the APFD metric in most cases. Therefore, it is safe to use a higher value of *stp* in hope of getting a better running time.

Table 6.7: Average Running-time (in sec), Stage 4.

	ant	***jmeter***	***xml-security***	***nanoxml***	***galileo***
1	496	3.3	1.2	6	66
5	98	0.5	0.8	3	16
10-a	138	0.5	0.9	3	26
10	54	0.5	0.7	2	10
20-a	95	0.5	0.8	3	16

The primary impact of the chosen *stp* values is on the running-times, as shown is Table 6.7. As the *stp* parameter increases, the running-times decrease linearly. In the case of *adjustive* parameters, note that, they should be counted on the average about one half of their specified values. Considering that APFD values are similar for different *stp* values, using larger numbers for this parameter can improve the cost effectiveness. That said, note that the involved *stp* values in this study are relatively low. If we choose very large values, the BNA technique will lose its feedback characteristic. Even with the current *stp* parameters, it can be observed in Figure 6.1 that lower values result in better APFD (although not statistically significant). Naturally, if we keep increasing the *stp*, at some point the APFD will become significantly lower, as it does for *ant* object. This suggest that this parameter should be selected carefully and using similar experiments.

6.3 Discussion

The reported results indicate that the parameters can, in some cases, impact the performance of the proposed framework. In other cases, however, the choice of the parameters is not important and hence those parameters can be chosen with less cautious. Experiments such as this can help the selection of the parameters. Regarding the research questions, lessens could be learnt from the obtained results.

Regarding the first research question, impacts of using feedback mechanism, the obtained results present an interesting pattern. While using feedback results in faster detection of faults for TSL-based test suits, for Junit test suites, it does not affect the performance significantly. This could be attributed to different natures of test cases among these two families of test suites. Junit test developers are encouraged to target one and only one functionality of the system in each test case, and hence, different test cases are less likely to cover similar parts of the code. In TSL, for one specific operation, multiple test cases with different parameters could be generated, and hence, a larger amount of redundancy can be potentially present. This is reflected in the large number of available test cases for the TSL objects, compared to JUnit suites. The feedback mechanism, obviously, is of use only when there are test cases which cover same parts of code. If the test cases did not intersect in any of the covered code, they use of feedback would not change the algorithm. Since in JUnit test cases are less likely to cover common parts, the use of feedback for JUnit test suits does not improve the performance as much as it does for TSL. The feedback mechanism, however, increases the (machine) running-time always, and consequently, if this factor matters, testers need to be more careful about the feedback mechanism.

Regarding the second research question, effects of different measurement techniques, it appears that the alternative approaches are quite similar. Specifically, in the case of change information, the differences are never significant. This could be due to the fact that all those techniques extract similar information from the system. The fact that each of those sources of information builds only one part of the whole model also contributes to their low impact. Nevertheless, the fault-proneness models were occasionally influential, and hence, it can be beneficial to investigate which model fits a specific program in the early phases of a project. Furthermore, there is a handful of other fault-proneness models and metrics that were not investigated in this work. Most importantly, the faults used in this experiment are all mutation faults. The process by which mutants are created does not take into the account the quality of the code. Hence, the distribution of faults may be different from that of a real system, and as a result the quality metrics may lose their prediction capability for faulty code. This suggest future experiments should focus on real faults as opposed to mutation faults.

Regarding the third research question, the effects of different parameters, the running time can be considerably affected. In practice, it is crucial to find the inference algorithm that suits a particular system. Based on the observed data and mentioned justifications, when dealing with large systems which have a large number of classes, exact algorithms are not recommended. The results also indicate that performing inference for each single test case when using feedback is not needed. Adding several test cases after each inference can accelerate the technique while affecting the fault detection rate only slightly.

Finally, the obtained results suggest that the impact of one particular variable can differ from one system to another; therefore, in practice, one can use a similar methodology to investigate the impact of different variables in their specific domains. The fact that the influential and non-influential variables are spotted in this experiment can help such investigations to focus on the important factors. The existence of many non-influential variables is a valuable finding of this study.

Chapter 7

Experiment 2: Evaluation Using Mutation Faults

The performance of the BN-based framework needs to be compared against existing approaches from the literature. As mentioned before, the code-based realization is a good candidate for such comparison. This experiment is designed to compare the code-based realization to conventional coverage-based techniques from the literature through an empirical study.

As explained in Section 5.3, artificially generated faults need to be utilized for the purpose of such studies. Two types of mutation and hand-seeded faults are available for SIR objects. This experiment reports on an experiment that utilizes mutation faults to empirically evaluate the performance of the proposed framework. The following research questions are investigated in this experiment.

- **RQ1:** Does the proposed code-based realization of the BN-based framework result in faster fault detection?
- **RQ2:** Does the use of feedback mechanism enhance the rate of fault detection?

The first question is the main focus of this experiment. The second question has been asked in the previous experiment as well but because of the importance of this factor, it is included here again. This chapter reports on a controlled experiment designed and performed to answer these questions. The rest of this chapter first describes the experiment design, including variables and measures and then explicates threats to the validity of the experiment. Obtained results follow next and at the end, a discussion on the obtained results is presented.

7.1 Experiment Design

In a control experiment, one needs to identify the independent variables that affect the system. The effects of the independent variables are evaluated based on the dependent variables of interest. The following details the involved independent and dependent variables.

7.1.1 Variables and Measures

Independent Variable: Involved Techniques

The aforementioned research questions concern the performance of the proposed framework versus that of existing techniques. Therefore, the independent variable of our study is the involved techniques. Five different techniques are included in this experiment: BN, BNA, MC, MCA, orig. These techniques are each described in more detail in Section 5.4.

As mentioned in Section 5.4, conventional coverage-based techniques are chosen for this experiment mainly because of their predominance in the literature. Moreover, they are the most precisely presented approach and hence replicating them is a doable task. Most other techniques from the literature, too, compare their results against this family of techniques. The assumptions of the coverage-based methods regarding the process of regression testing, etc. is similar to those of the proposed framework and hence a direct comparison is possible.

As for the BN family of techniques, it has been mentioned in the first experiment Section 6 that there exist parameters that can be used to adjust the performance of the system. In this experiment, the value of those parameters is not part of the research questions. Nonetheless, a particular configuration is required for an implementation of the technique to work. Experiment 1 has investigated the effects of those parameters. The results from that experiment has been used to come up with a good configuration for each object of study. Table 7.1 details the used configuration for all objects and parameters.

Table 7.1: Parameter Configuration for the BN-based Techniques.

Parameter	**ant**	**jmeter**	**xml-security**	**nanoxml**	**galileo**
Change Analysis	DIFF				
Quality Metric	DIT	WMC	RFC	WMC	DIT
Inference Alg.	Sampling	Exact			
stp	adjustive 10				

The impact of change analysis techniques has been found to be insignificant in the previous experiment. Hence, the simpler of the two approaches, namely DIFF, is chosen here for all the objects. In the case of quality metrics, since some instances of significant difference were indeed observed, here, for each object, the metric that produced the best results in Experiment 1 (according to the median values) is used. For the inference algorithm, based on the results obtained in Experiment 1, sampling has been chosen only for *ant* and exact for the rest. This is due to the fact that, except for the *ant* object, the exact algorithms are found to be faster. For the *stp* parameter, the *adjustive* 10 value is never significantly outperformed by any other value. At the same time, it provides a relatively good running time. This value has been hence used throughout this experiment for all objects. Note that values of most these parameters have been found to be insignificant as far as early fault detection is concerned and hence this particular choice of configuration is unlikely to impact the results.

Dependent Variables

APFD metric is an indicator of the rate of fault detection, the subject of the research questions in this experiment. This metric is hence used throughout this experiment as the dependent variable. For each independent variable (technique), on each object of study, the value of APFD metric is computed through the process described in Section 5.1 and using mutation faults. Boxplot diagrams of APFD metric for each single version is depicted and an overall (over all releases) statistical analysis is performed to find the significance of observed results. The boxplots and statistical analysis is based on a distribution of 30 different runs. For each version of each object, from the pool of available mutants, 30 set of mutants are randomly chosen, each of which has a random size between 1 and 10. For each mutant set, a different APFD value is obtained. Therefore, for each version of each object 30 different runs are available.

7.1.2 Threats to Validity

This section describes the construct, internal, and external threats to the validity of this experiment as well as the cautions taken to limit the effects of the threats. Because the set of experiment objects and faults is similar between this experiment and those of Experiment 1, many of the threats to the validity are common.

Construct Validity. The dependent measures used in this experiment are not complete in terms of capturing all the facets of early fault detection. Other metrics could be defined for measuring the rate of fault detection accounting for other factors such as time required by each test case. The APFD metric threats all test cases identically but this is not always the case in real systems. However, the test cases of the objects under study do not vary drastically in the execution time and hence the assumptions of APFD does hold for the most part.

Internal Validity. Several other factors could influence the results and affect the inferences made from the results. Like Experiment 1, the tools developed/used to collect data can potentially have defects. Moreover, the choice of the configuration values for the involved parameters can potentially influence the results. Small investigation have been made prior to the study as to the effects of the parameters; those informal studies indicate that the results are not affected significantly. The formal results presented in Experiment 1 also has revealed that the code-based realization is not much sensitive to the particular values of the involved parameters as far as the rate of fault detection is concerned.

External Validity. There are also threats to the external validity of the experiment, similar to Experiment 1. The programs studied here are relatively small to medium size open source software. Complex industrial programs may be subject to different concerns. Same applies to the underlying testing process. The use of mutation faults is again an important threat to generalization of the results. Mutation faults do not follow the distributions assumed by the fault-proneness models and hence the power of BN-based techniques is partially reduced. Although for these reasons using real faults is preferable, those are typically not easily available for research purposes. It has been shown that mutation faults can be representative of real faults [35], although the result from such studies should be interpreted in the light of the fact that not all measures of representativeness are studied. Finally, this experiment compares only two variations of the existing coverage-based techniques; other variations can be compared differently. Nevertheless, previous studies (*e.g.,* [39]) have not revealed a major and consistent difference in the performance of the alternatives.

7.2 Obtained Results

For each object of the study, the obtained results is separately given in the following sections to enhance a better presentation. The reported results, for each object, consist of a boxplot diagram depicting the APFD metric for each version of the object and also for all the releases put together. In addition, a table lists the results from Kruskal-wallis statistical test in the form of pair-wise comparisons. The pairs of techniques are compared against each other using the statistical test.

7.2.1 *ant*

Figure 7.1 depicts the APFD metric as the dependent variable of this experiment for each version of the *ant* object. A result tagged as release *v*, represents the the regression testing of version *v* of the program. For the first version, release 0, no results are available because the the evaluation procedure relies on the coverage data from a previous version which does not exist for the very first version. The last boxplot diagram shows the overall result for all the releases.

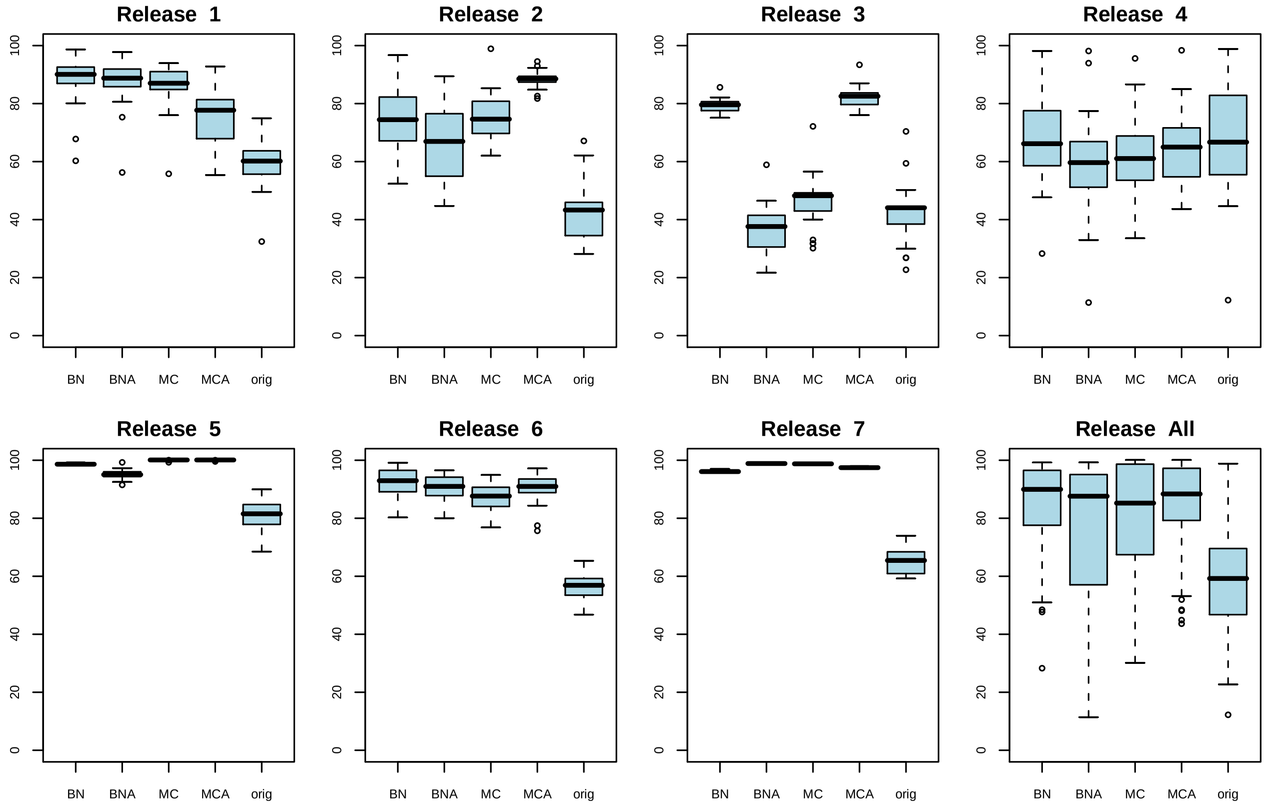


Figure 7.1: The APFD Metric for *ant* using Mutation Faults

It is evident from the boxplot diagram that all four heuristic techniques result in faster fault detection compared to the original ordering in most cases. Comparing the four heuristic techniques, however, the results vary from a release to another. Overall, BN technique seems to slightly outperform BNA and MC. Also, MCA seems to outperform BNA and MC. These two (BN and MCA), however, seem to give results close to each other.

Kruskal-Wallis test can help further analyze the data and understand the relative performance of the techniques. Table 7.2 shows the result of the statistical test for the *ant* object. The same format will be used for the following objects as well. Each row in these tables shows the results of comparing a pair of techniques. There are four lines in the tables. The first line compares APFD values of BN versus that of BNA. Such comparison helps address RQ2, whether the use of feedback improves the performance of the technique. The second and third rows correspond to RQ1, whether the BN-based framework is performing better than conventional coverage-based approaches. Notice that non-feedback variations of the two techniques are only compared against each other (BN vs MC) and so are feedback-employing versions (BNA vs MCA). This way, the effect of feedback is studied separately. The last row in the table is the combination of comparing the control *orig* technique versus each of the other four techniques, all put together to save on space. The details of the four comparisons are not that interesting because they all indicate only one thing: all the heuristic techniques are outperforming the control technique.

Table 7.2: Kruskal-Wallis Test Results for *ant* Object.

Compared Techniques	**p-value**	**chi-squared**
BN vs BNA	0.00287	8.889
BN vs MC	0.18420	1.763
BNA vs MCA	0.00212	9.437
orig vs others	< 0.0001	> 60

Interpreting the results of the statistical tests presented in 7.2 for a 0.1% level of confidence, there would be no significant difference between the the coverage-based and BN-based techniques. Nor would the use of feedback make any difference for that level of significance. If aimed at 1% level, the results from first and third rows are statistically significant. Whether statistically significant or not, the fact remains that BN is outperforming BNA overall, suggesting that the use of feedback is not helpful here. Also, the performance of BN-based and coverage-based families of techniques is too close to draw a meaningful distinction. It could be said that the two seem to perform comparably for *ant* object.

7.2.2 *jmeter*

Figure 7.2 depicts the APFD metric for each version of the *ant* object. The last boxplot diagram shows the overall result for all the releases.

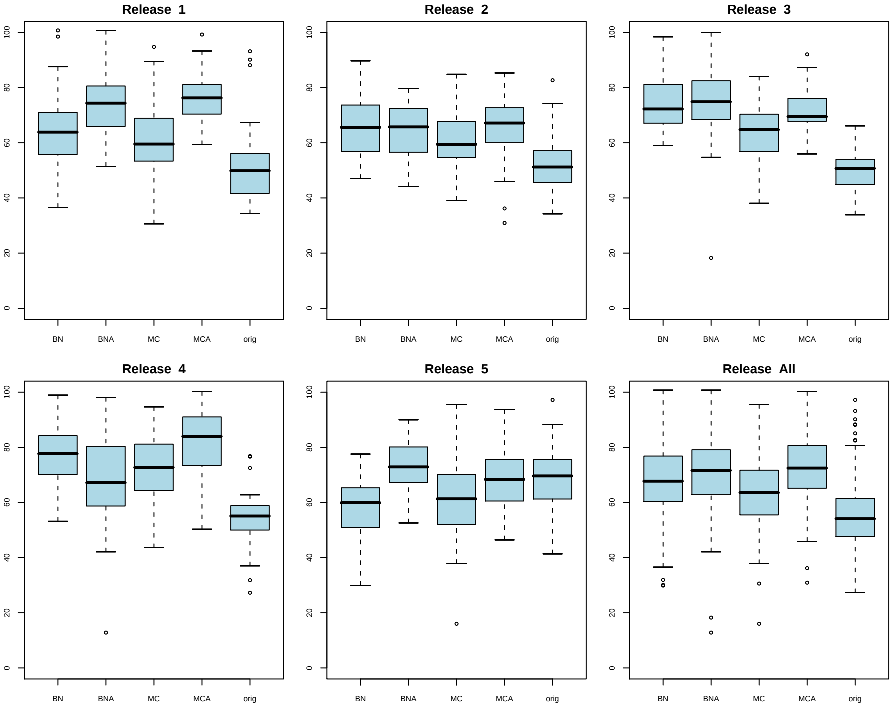


Figure 7.2: The APFD Metric for *jmeter* using Mutation Faults

On this object, also, all four heuristic techniques result in a rate of fault detection better than that of the original ordering in most cases (all releases except version 5). Again, among the four heuristic-based techniques, the results vary from a release to another. In some releases such as 1 and 2, the APFD values are comparable. In others such as 5 the BN family works better, in others such as version 4, coverage-based family does better. Overall, the BNA technique seems to slightly outperform BN; the BN appears to be better than the MC; and the BNA and MCA seem to be very close. The further analyze of these observations is possible through the Kruskal-Wallis test.

Table 7.3 shows the result of the statistical test for the *jmeter* object. A same format as that of the previous table is used here. Again, the details of the four comparison for *orig* are not mentioned to save space. Those tests again indicate that all the the heuristic techniques are outperforming the control *orig*.

For this object, similar to *ant*, the interpretation of the results depends on the selected level of significance. For a 0.1% level of confidence, there is no significant difference between the the coverage-based and BN-based techniques; nor is there between BN and BNA. If aimed at 1% level, the results from first and second rows are statistically significant. In that case, BNA is outperforming the BN which in

Table 7.3: Kruskal-Wallis Test Results for *jmeter* Object.

Compared Techniques	**p-value**	**chi-squared**
BN vs BNA	0.04415	4.050
BN vs MC	0.00598	7.555
BNA vs MCA	0.22577	1.467
orig vs others	< 0.0001	> 35

turn is better than MC. The use of feedback seems to be quite effective here but not as much as one would expect. The non-feedback version of the BN-based framework seems to find faults faster than non-feedback version of coverage-based approach but there is no real distinction between the feedback-employing counterparts.

7.2.3 *xml-security*

Figure 7.3 illustrates the obtained APFD metric as the dependent variable of this experiment for each version of the *xml-security* object. The last boxplot diagram shows the overall result for all the releases all put together.

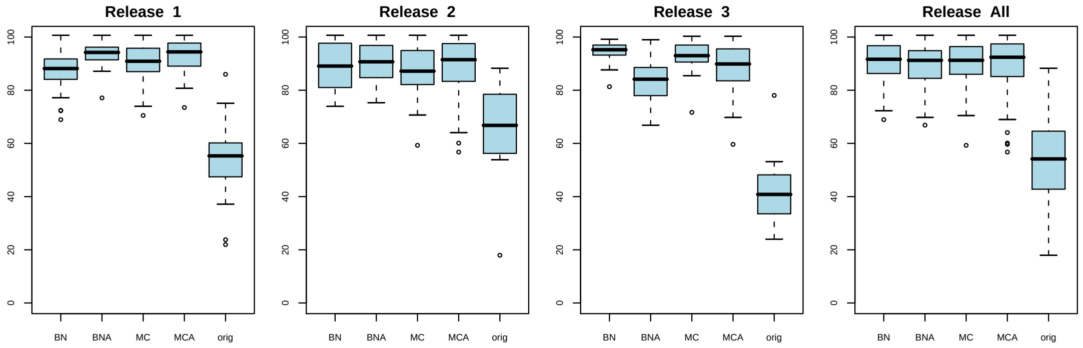


Figure 7.3: The APFD Metric for *xml-security* using Mutation Faults

All the four heuristic-based techniques are much faster than the original ordering in finding bugs and in all the three versions. As with the previous two objects, the four heuristic-based techniques vary in relative performance from a release to another. In version 1, both feedback techniques BNA and MCA seem to perform better. In version 2, there is no real difference. In version 3, non-feedback techniques, specially BN is performing better. Overall, the APFD values of the four techniques are close and no real distinction can be claimed.

To further evaluate the observations, Kruskal-Wallis test is performed. Table 7.4 shows the result of the test for the overall result on *xml-security* object. The table follows the same format as the previous two. Again, the details of comparing *orig*

Table 7.4: Kruskal-Wallis Test Results for *xml-security* Object.

Compared Techniques	**p-value**	**chi-squared**
BN vs BNA	0.23397	1.417
BN vs MC	0.82786	0.0473
BNA vs MCA	0.10911	2.6567
orig vs others	< 0.0001	> 120

against other four techniqes are not mentioned. Those tests, again, indicate that all the heuristic techniques are outperforming the control *orig*.

The results from Table 7.4 indicate there exist no statistically significant difference between the compared pairs of techniques. This finding is true for both 0.1% and 1% levels of confidence. This confirms that on this particular object, first the use of feedback does not make any improvement; second, the two families of techniques achieve similar performances. Note that on this relatively small object, the performance of all four techniques is remarkably good. The achieved APFD values are around 90% in average and the boxplots show that there is comparatively little variance in the data. To increase the performance on an object like this requires very comprehensive models.

7.2.4 *nanoxml*

Figure 7.4 depicts the APFD as the dependent variable of this experiment for each version of the *nano-xml* object. The last boxplot diagram shows the overall result for all the releases all put together. It should be mentioned that the version 4 of this object has only one mutated version. Therefore, it is not possible to generate 30 sequences of mutant groups for it. The depicted results represents only one instance (with that one mutant) and hence does not provide a reliable source of comparison. Version 4 is taken out of discussion for this object.

Resembling the pattern observed on the previous three objects, all four heuristic techniques have much better APFD distribution than that of the original ordering. For this object however, compared to the previous three, the relative performance of the heuristics follows a much more regular pattern. The two feedback-employing techniques seem to outperform their counterparts most of the times (except for the MCA on version 1). The MC technique seems to be better than BN most of the times. The performance of the BNA and the MCA are similar for the most part. Overall, BNA seems to achieve the best performance among the four.

To further analyze the data, Kruskal-Wallis statistical test is conducted. Table 7.5 shows the result of the test for the *jmeter* object. A same format as that of the previous tables is used here. Once more, the details of the four comparisons for *orig* are all summarized in the last row.

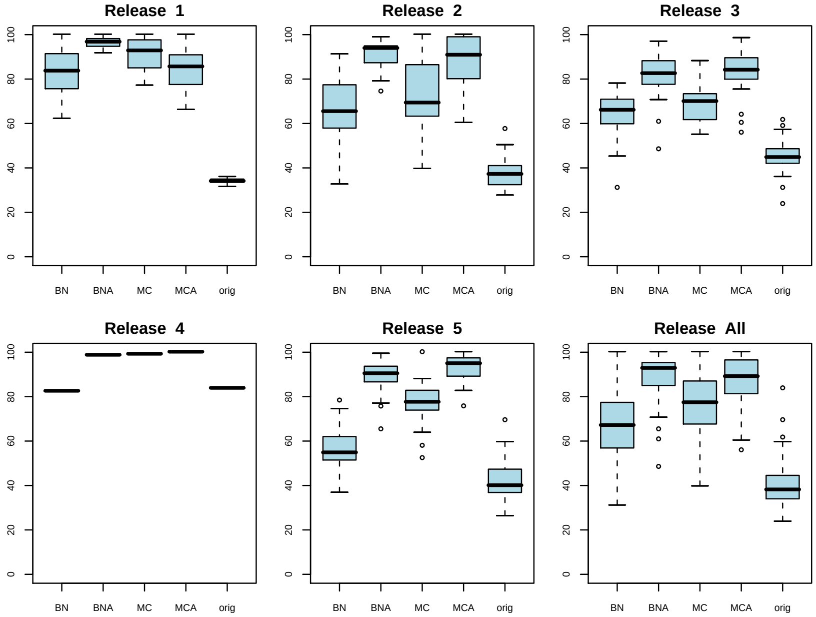


Figure 7.4: The APFD Metric for *nanoxml* using Mutation Faults

Table 7.5 confirms that for *nanoxml* object more rigorous observations can be made, compared to the previous cases. Here, for both significance levels of 0.1% and 1%, significant results are obtained. The BNA technique is significantly better than BN, revealing that the use of feedback has been effective here. Comparing the BN and the MC, the coverage-based method is significantly outperforming the BN-based techniques on this object. On the feedback-employing versions, on the other hand, the BN-based techniques are performing better, although not significantly according to the test.

7.2.5 *galileo*

Figure 7.5 portrays the APFD measure as the dependent variable for each version of the *galileo* object. The last boxplot diagram shows the overall result for all the 15 available releases. As the Table 5.1 indicates, this object is the richest among the five in terms of the number of available versions, test cases, and mutants.

Table 7.5: Kruskal-Wallis Test Results for *nanoxml* Object.

Compared Techniques	p-value	chi-squared
BN vs BNA	< 0.0001	111.02
BN vs MC	< 0.0001	25.08
BNA vs MCA	0.16112	1.96
orig vs others	< 0.0001	> 140

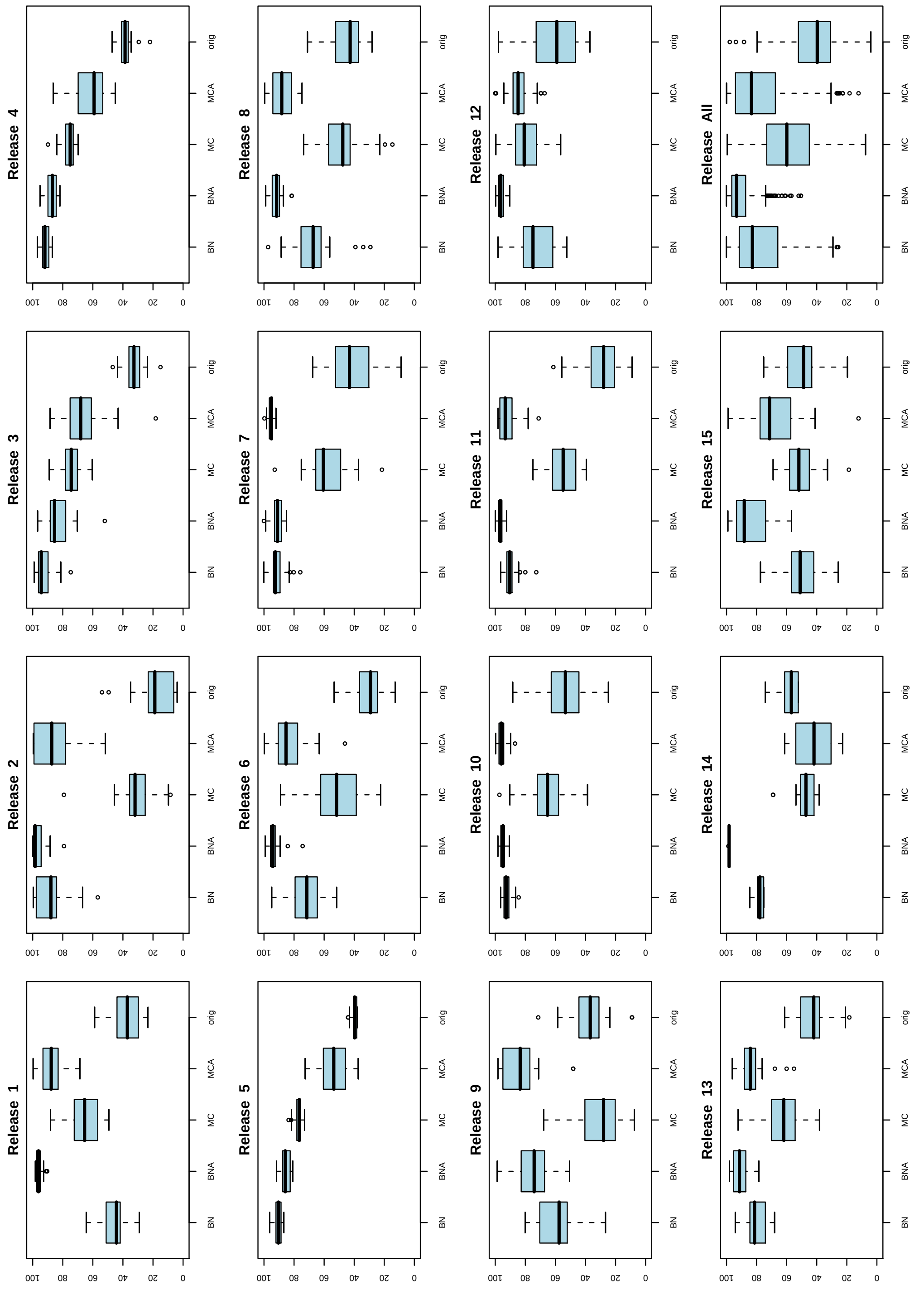


Figure 7.5: The APFD Metric for *galileo* using Mutation Faults

The control *original* technique is again surpassed by the heuristic techniques in most cases (MC on version 9, and MC and MCA on 14 are the only exceptions). Specially, no BN-based technique is ever been outperformed by the *orig* technique. In comparing the heuristics against each other, a very clear pattern exists for this object. The two BN-based techniques are doing better than their coverage-based corresponding techniques on most of the versions. The BN specially seems to be often on large distances from MCA. Between the two BN-based techniques, the feedback-employing version has the upper hand in most cases (only in three instances this is not true). Overall, BNA is producing better results than both BN and MCA, with BN outperforming MC. The initial observations on this object suggests the advantage of BN-based techniques over coverage-based and also the benefit of employing feedback.

To further analyze the data, Kruskal-Wallis statistical test is conducted. Table 7.6 shows the result of the test for the *galileo* object. The format of this table is same as that of the previous tables. The details of the four comparison for *orig* are again omitted for the same reasons.

Table 7.6: Kruskal-Wallis Test Results for *galileo* Object.

Compared Techniques	**p-value**	**chi-squared**
BN vs BNA	< 0.0001	182
BN vs MC	< 0.0001	216
BNA vs MCA	< 0.0001	123
orig vs others	< 0.0001	> 188

The results of the statistical test confirm the informal observations. For this object, it could be rigourously asserted that BN-based techniques achieve a faster fault detection rate compared to coverage-based techniques. It could be also asserted that the feedback mechanism increases the performance significantly. All the differences are statistically significant. The average performance of the BNA technique is above 90%, a more than 10% increase from the existing MCA.

7.3 Discussion

The obtained results help answering the two research questions asked at the beginning of this experiment. RQ2 has been also discussed in the Experiment 1. The results from this experiment confirm the findings from the first experiment: feedback seems to be of value for TSL test suites but does not add much to JUnit objects.

Regarding RQ1, the results indicate the proposed code-based realization of the BN-based framework in some cases outperforms the existing coverage-based techniques and performs comparably in other cases. This conclusion is in line with the previous research in this area that assert various techniques are suitable for different environments. The fact that the BN-based framework can incorporate many heuristics into one model, the author claims, makes it a more reliable technique when considered across a variety of environments. As evidence for this claim, consider that the BNA technique is never significantly dominated by other techniques (over all versions) for the studied objects whereas MCA is.

The claim of more stability for the BN-based framework is supported by the obtained results when seen as a whole. Table 7.7 reports the average and variance of all the obtained APFD values across all the objects. These numbers demonstrate how the BN-based framework is more stable across different objects. The two BN-based techniques result in both higher averages and lower variances compared to their similar coverage-based counterpart. BNA, in particular, achieves highest average and lowest variance among all the techniques. This relative stability of the proposed framework is due to the fact that it utilizes many heuristics simultaneously. Although, having a detailed knowledge of a particular system, one might be able to find heuristics that suit the system, in the typical absence of such knowledge, using many heuristics is a safer approach. This way, for any object, some heuristics ensure that decent results are obtained.

Table 7.7: Average APFD Results Across All Objects.

	BN	**BNA**	**MC**	**MCA**	***orig***
Average	77.6	84.4	68.5	80.9	48.0
Variance	16.8	15.6	20.3	16.0	16.9

Another interesting pattern can be observed in this experiment. In objects with TSL test suites, the differences between the performance of various techniques are frequently significant; in objects with JUnit suites it is not so. This is most probably because of the different natures these two testing technologies have. Looking back at Table 5.1, TSL objects have much more number of test cases compared to JUnit objects considering the size of the programs. This is due to the more fine-grained nature of the TSL test cases. As such, the coverage data for these objects are more fine-grained and hence more descriptive. On such fine-grained data, the distinctions between the approaches is further highlighted. In JUnit test suites, in the absence of precise low-level data, all the techniques perform comparatively.

Finally, there are important issues that need to be considered when interpreting the results. First, the coverage data used here, for the reasons mentioned in Section 5.4, are in different levels of granularity between the two family of approaches. The fact that coverage-based techniques use method level coverage data gives them an edge which is not inherent in those approach itself. Similar experiments have been performed with coverage-based techniques abstracted to class-level; the difference between BN-based and Coverage-based approaches are greater in that case. Second, note that the fault-proneness models used in the BN-based techniques have lost their prediction power to some degree in this study because of the use of mutation faults. In systems with real faults, the BN-based techniques are likely to perform better. Evaluating the system using hand-seeded faults which are in general believed to be more representative of the real faults can help answer such concerns.

Chapter 8

Experiment 3: Evaluation Using Hand-seeded Faults

The reported experiments on the mutation faults has revealed the relative advantages of the BN-based techniques. There are, however, notable concerns with the representativeness of mutation faults. For instance, mutation faults are distributed randomly across software but real faults tend to reside more often in low quality sections of the code. The deficiencies of mutation faults reduce the heuristic power of the BN techniques. It is conjectured that with real faults BN would perform better than with mutation faults. To evaluate that claim, a third experiment is performed that uses hand-seeded faults instead of mutants. As discussed in Section 5.3, hand-seeded faults are believed to be more characteristic of real faults; here, they have been chosen in the absence of real faults for the objects under study.

This chapter reports on the experiment performed with hand-seeded faults provided for the SIR objects. The research questions pursued through this experiment are similar to those of Experiment 2, only investigated on a different type of faults:

- **RQ1:** Does the proposed code-based realization of the BN-based framework result in faster fault detection?
- **RQ2:** Does the use of feedback mechanism enhance the rate of fault detection?

The rest of this chapter, first describes the design of this experiment; then, it explicates threats to the validity of the experiment; obtained results follow next and at the end, a discussion on the results is presented.

8.1 Experiment Design

This experiment uses hand-seeded fault available for SIR objects for the evaluation purposes. It should be noted that hand-seeded faults do not exist for the *galileo* object. On three other objects, namely *jmeter*, *nanoxml*, and *xml-security*, the number of seeded faults are minimal, one or two for each version. Considering that those objects have only a few versions available, there is little ground for experimentation on those objects. This experiment, hence, considers only the *ant* object which has more versions and also occasionally more faults. The volume of available data in this experiment is not sufficient for statistical tests and therefore informal analysis need to be relied upon. As such, this experiment serves as a proof of concept rather than a control experiment.

Table 8.1 details for each version of the *ant* object the number of available faults, test cases, and also the size of the program in terms of the number of classes. Comparing this table and that of 5.1, one notices the difference between the number of available test cases. This is due to the fact that here the meaning of a test case is different than that of the previous two studies. For the JUnit test suites, the fault matrices available in SIR are constructed differently for hand-seeded and mutation faults. For the hand-seeded faults, a test case is defined as one test class; for mutation faults, each method of each test class is treated as one test case[1]. This results in less number of available test cases for hand-seeded faults.

Table 8.1: The Properties of Eight Consecutive Versions of Apache Ant.

Metric Name	**ver. 0**	**ver. 1**	**ver. 2**	**ver. 3**	**ver. 4**	**ver. 5**	**ver. 6**	**ver. 7**
Faults Count	0	1	1	2	4	4	1	6
Test Case Count	0	28	34	52	52	101	104	105
Number of Classes	143	229	343	343	533	537	537	627

Because of the limited number of available faults, for each version only one evaluation is performed in which all available faults are included. This is different from the previous experiments in which several sequences of fault sets were randomly chosen from the pool of available mutants. The setup of this case study, otherwise, is similar to that of Experiment 2. The techniques are the same, with the same configuration and implementation. The metric of evaluation is again APFD, which measures the rate of fault detection.

8.1.1 Threats to Validity

This section describes the construct, internal, and external threats to the validity of this experiment as well as the cautions taken to limit the effects of the threats. Some of these threats are those that were present in previous experiments as well.

Construct Validity. The dependent measures used in this experiment are not complete in terms of capturing all the facets of early fault detection. Other metrics

[1] The latter makes more sense according the definition of a test case in JUnit.

could be defined for measuring the rate of fault detection accounting for other factors such as time required by each test case. Moreover, a metric such as APFD is most revealing when a noticeable number of faults exists. In some versions of this study that is not the case.

Internal Validity. Several other factors could influence the results and affect the inferences made from the results. Similar to the previous experiments, the tools developed/used to collect data and their configuration can potentially have defects. Furthermore, the number of available faults were not enough to conduct statistical tests and hence the observations are only informally analyzed. This reduces the confidence in the inferences made according to the results.

External Validity. There are also threats to the external validity of this experiment. The program studied here is of medium size. Complex industrial programs and other processes may be subject to different concerns. The use of hand-seeded faults, by itself, is a threat to generalization of the results. These faults, although believed to be more representative than mutation faults, do not necessarily follow the distributions assumed by the fault-proneness models. Although for these reasons using real faults is preferable, those are typically not easily available for research purposes. Even more importantly, the number of available faults is limited here. Real software systems vary in the fault content. Those which contain more faults could be subject to other concerns.

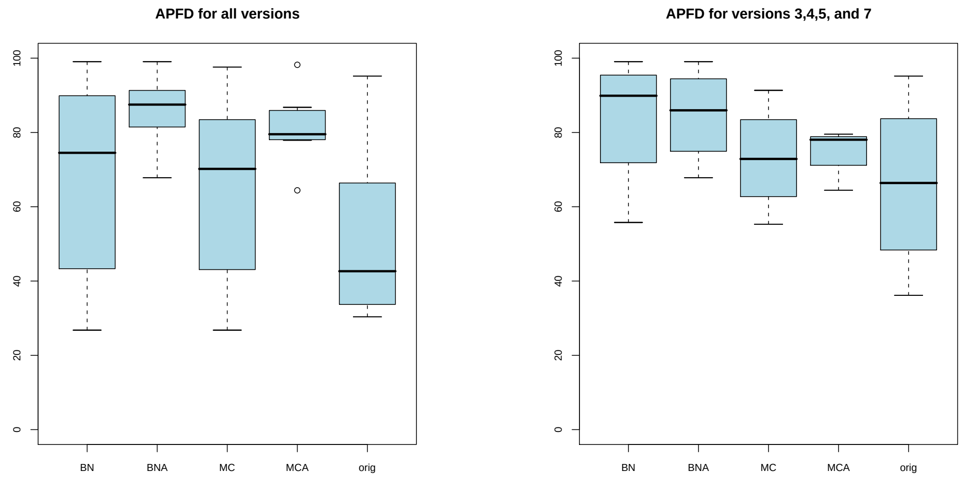
8.2 Obtained Results

The obtained results for different techniques on the *ant* object is depicted in Figure 8.1, part A. Each box represent only 7 figures, each figure corresponding to the APFD metric achieved on one of the 7 releases. Also, Table 8.2 reports the average and the standard deviation of the APFD values achieved by each techniques.

Table 8.2: APFD Statistics of Different Techniques on Hand-seed Faults for *ant*

Technique	**All Versions**		**More Than One Fault Versions (3,4,5,7)**	
	Average	SD	Average	SD
BN	66.68045	29.39683	83.64397	19.143807
BNA	85.70182	10.04899	84.68548	13.229761
MC	63.94817	27.70506	73.09324	14.880748
MCA	81.44635	10.32285	75.01233	7.094856
orig	52.62632	24.43825	66.03252	24.576512

According to the boxplot diagram and the table, all the four heuristic techniques perform better than *orig*, the control technique, by far. The average APFD of *orig*



(A) All versions

(B) Versions 3,4,5, and 7.

Figure 8.1: Boxplot Diagrams of the APFD Results for Hand-seeded Faults on *ant*.

is 11% lower than second worst technique, the MC, and 33% lower than the best technique, the BNA. This demonstrates the advantage of all heuristics in this case.

Regarding RQ2, the two feedback-employing techniques seem to top the non-feedback techniques both according to the median of the data and the variance. The BNA and MCA techniques are in average around 20% faster than their non-feedback variations in finding hand-seeded faults. They also have around 20% less deviation which means they are more stable in performance. Note that this results is in contrast with those observed on mutation faults. In second experiment presented in Chapter 7, on *ant* object, the use of feedback has not caused any significant improvement. This could be attributed to many factors. First, the definition of what a test case is differs in these two experiment; that could have affected the results. Second, the difference in the nature of faults could have resulted in this dissimilarity. Finally, it is possible that the observation in this experiment, which is not confirmed by any statistical test, is arbitrary. Finding the real cause calls for further experiments with a larger number of hand-seeded or real faults.

As for the two families of techniques, BN-based techniques seem to result in slightly better APFD values compared to coverage-based techniques. The BN and BNA techniques are in average respectively 3% and 4% better than the MC and MCA. The deviations are comparable between the two families. Overall, the differences between the two families are hardly significant. Taking a closer look at the data, in many versions, BN-based techniques are achieving significantly better results. To analyze the data further, the fault content of the subject program should be considered. There are three versions of the *ant* that are seeded only with one fault. The results indicate that on these three versions, BN-based techniques achieve lower fault detection rates. One can argue that one single fault does not provide a reliable basis for comparison of techniques. Thus, we took out the versions with one fault (versions 1, 2, and 6) and compared the results again.

Figure 8.1, part B illustrates that APFD values achieved only on versions 3,4,5, and 7 (versions that are seeded with more than one fault). Also the second column of Table 8.1 show the average and standard deviation for those versions. The results show that the difference between the performance of the BN-based and coverage-based techniques is much more significant, in this scenario. The BN and BNA techniques are in average 11% and 10% better than the MC and MCA, respectively. Note that the performance of BN and BNA techniques are very close in this case, suggesting that the pattern observed in the Experiments 1 and 2 is more reliable than those seen in the first part of this experiment when all versions are considered.

8.3 Discussion

Observations could be made from the obtained results regarding both research questions.

Regarding the comparison of feedback-employing and non-feedback techniques, mixed results are obtained when considering all versions or only those with more than one fault. To further inspect this phenomenon, we depicted APFD of the techniques versus the number of faults. Figure 8.2 shows the regression lines of such data.

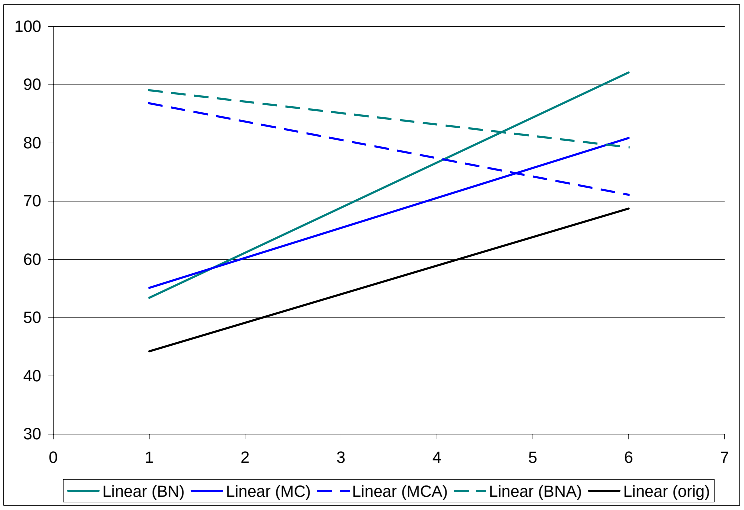


Figure 8.2: The Regression Line of the APFD vs. Fault Content

This figure indicates that as the fault count grows, the APFD values of feedback techniques decrease, whereas those of non-feedback techniques increase (both feedback techniques have a negative slope but non-feedback techniques have positive slopes). The original technique, too, has a positive slope suggesting that with more faults, better APFD values are indeed expected. The decreasing slope of feedback techniques suggests either the cases with one fault are not reliable enough for the comparison or that the power of feedback mechanism degrades when more faults are present. To find the real effect of fault content, empirical studies and controlled experiments are needed. If the observation of this experiment is generalized, it has a very important practical implication: when the software is believed to contain many faults, the use of feedback is not useful but in more reliable systems, when testers struggle to find the last faults, feedback can improve the rate of fault detection.

The results from this experiment have shown the BN-based techniques to achieve better rate of fault detection compared to coverage-based techniques. This suggests that where experiments using mutation faults did not show any significant difference between BN-based and coverage-based techniques, there could be differences that can be revealed by using other types of faults. On the *ant* object, in this experiment, the BN-based techniques outperformed coverage-based and if considering only versions with more than one fault, significantly so. This is in contrast to the results obtained in the previous experiments. Nevertheless, the data available for this experiment are relatively sparse and hence the results should be interpreted with care.

Chapter 9

Conclusions and Future Directions

This thesis has proposed a new BN-based framework for the regression test case optimization problem and evaluated its performance. The following contributions have been made.

- The thesis has proposed a novel framework for solving the regression test optimization problem. The framework predicts the probability of test cases finding faults using multiple heuristics.
  - Proposed BN-based framework *i*) provides flexibility in the utilized measurements, and *ii*) enables combining different heuristics together based on their rational relations rather than ad-hoc methods.
  - The BN-based framework is presented in a general sense with minimum assumptions about available measurements.
  - A path to the realization of the general framework for specific environments is provided.
- The thesis has presented a special code-based realization of the general BN-based framework. Software code is the only artifact this realization uses for measurements. This makes the realization applicable for most software systems.
  - The code-based realization heuristically utilizes three measurement techniques to model the regression fault detection event: code change analysis, fault-proneness modeling based on code structural quality metrics, and test code coverage gathering.
  - The realization is implemented to get a working regression test optimization tool, focusing on test case prioritization.
  - The realization is in line with most of existing research in this area in terms of assumptions and used sources of information. This facilitates experimental comparison.

- Three sets of experiments have been conducted to empirically evaluate the approach and the results are reported.
  - The first experiment investigates the effects of parameters present in the implementation of the framework on its performance. The results indicate little sensitivity in terms of fault detection but significant differences in running-time. Also, feedback improves the performance significantly at least for some objects.
  - The second and third experiments compare the performance of BN-based framework against existing techniques from the literature on mutation and hand-seeded faults respectively. The second experiment reveals that on some objects BN-based techniques find bugs faster but on the others perform comparably. The third experiment, however, suggests that even on those objects the performance of the BN-based techniques can be much higher if real (or more representative) faults exist in the objects.
  - Each experiment involves statistical analysis and critical discussion on the obtained results.

To wrap up the discussion, in this chapter, first a critical review of the presented research work is given. Based on the identified rooms for improvement, then, some directions for future work is described. Finally, a conclusion section sums up the arguments made in this dissertation.

9.1 Critical Review

The proposed BN-based framework introduces a new line of research in this area. As such, there are potential shortcomings in the presented work that need to be pointed out here.

The general BN-based framework- There are facets of the framework that the users, specially practitioners, might find inconvenient. Understanding the proposed framework requires some knowledge of probabilistic modeling and also software measurements. Since the framework is meant to be general and flexible, the users need to create realizations and hence do need to understand some details regarding how the framework works. This is the price to be paid for the provided flexibility. Moreover, the presented framework, like other approaches to the problem, does not provide a complete modeling of the system. It is, for example, merely based on estimating the current status of the system and history information is not taken into account. Because this framework is based on Bayesian concepts, it could be easily modified to account for such data.

Realization Diversity- Although other realizations of this framework have been developed, here, only one code-based realization is reported. Realizations that rely on artifacts other than code need to rely on information that are not easily available to researchers. This framework has been realized for requirement-based systems in an industrial setting [94] but the details of that line of research are not included here to avoid conflicts with the policies of the involved industrial partner.

Code-based Realization- The code-based realization, itself, has made simplifying assumptions that could affect the performance. Factors such as change impact could be easily incorporated. Nonetheless, more comprehensive models of the software system entail more costs and the cost-effectiveness of such models, when developed, need to be rigourously studied.

Empirical Experiments- Similar to any other research work, there are also threats to the validity of the empirical studies presented here. Each experiment has explicitly stated such threats and the inferences have been made with cautious. The representativeness of the involved techniques, faults, programs, measurements all could be argued. Although many precautions have been taken to reduce threats, more experiments are always needed to gain better confidence on the observations. Future research is needed to address issues like this.

9.2 Future Work

There are numerous ways to extend this research work. The general BN-based framework, its potential realizations, the code-based realization, and the empirical studies on the performance of the framework, can all be the subject of future research.

The General Framework: It can be enhanced by considering events that are not currently modelled. As an example, changes in software can be caused by other events such as debugging activities. Including those events can enhance the performance by providing a more comprehensive model of the system. As another example, in a continuous testing process, where regression testing is performed often, one can model the results from the previous test execution sessions as events that cause future changes (for bug fixes). That way, history information can be incorporated to the model. Such events are only indirectly modelled in the current framework; explicating them enables taking advantage of related measurement techniques.

More Realizations: Other realizations of the framework can be studied in the future. This thesis has presented one code-based realization, applicable in many cases, but to demonstrate its full capabilities further realizations for other settings are required. Those realizations can focus on other artifacts as the basis for modeling and measurement of software. Requirements, for instance, are the first class citizens in many software development and testing processes. Incorporating the potentially available information regarding requirements can improve the performance of the framework. Furthermore, if the traceability data between different artifacts is available in a project, multiple artifact types can be linked together to get a comprehensive model of the system. A realization can simultaneously exploit more than one type of artifact. In sum, the more realizations of the framework are developed in the future, the more its flexibility is demonstrated[1].

The Code-based Realization: The presented realization can be the subject of further enhancements. Most importantly, different techniques of change impact analysis could be incorporated easily. Such techniques are likely to increase the performance but are certain to impose more costs as well. The cost-effectiveness of incorporating change impact data, hence, needs further investigation. Moreover, this work has only explored class-level data. Other levels of granularity such as that of methods could be investigated. Again, important cost-benefit trade-offs exist and hence careful experimentation is needed. The most obvious way of enhancing the code-based systems is by enriching its set of measurement techniques. Demonstrating the potential advantages of other measurement techniques, however, calls for further empirical studies.

Elaborate Studies: Initial empirical studies have evaluated the performance of the proposed framework. More studies would help us better understand its pros and cons. Future studies can be conducted on a larger set of software systems, ideally with different natures. Large software systems from industry, the likely users of approaches such as this, are good candidates for such experimentation. Future studies can also investigate more variations of the proposed framework. For example, a limited number of fault-proneness models are investigated in this research work; more studies can follow to investigate other models. The most important aspect in future empirical studies is the nature of involved faults. This thesis, like most other academic research in this area, has used the artificially created faults. Since this framework makes assumptions about the distribution of faults, however, the artificiality of faults could potentially be important. Experiments with real faults are needed to understand the full power of the framework.

This research has produced promising results on an important line of research. In future, researchers can extend the work, as suggested, and evaluate its performance in more depth.

9.3 Conclusion

This dissertation has presented a novel framework for regression testing of software using Bayesian Networks (BN). The problem of software regression test optimization is targeted using a probabilistic approach. The framework models regression

[1] Note that there is ongoing research with industrial partners on the use of alternative realizations of the framework.

fault detection and as a set of random variables that interact through conditional dependencies. Software measurement techniques are used to quantify those interactions and Bayesian Networks are used to perform probabilistic inference on the distributions of those random variables. The inference gives the probability of each test case finding faults; this data can be then used to optimize the test suite for regression. Test case prioritization, for example, can be easily done by ordering test cases according to estimated probabilities of finding faults.

The proposed framework has two main goals: *i*) incorporating many sources of information in order to enhance performance, *ii*) providing flexibility in the used sources of information. The complexity of software renders comprehensive modeling infeasible, introducing a level of uncertainty. This is mitigated by measuring selected attributes of the system. We believe that using more sources of information enhances this process. Moreover, research in software regression testing has shown that dealing with different systems requires different strategies and hence flexibility is a desirable characteristic of any approach to the problem.

To provide flexibility, the proposed framework makes few assumptions about the software under analysis. Different realizations could be made to account for particular attributes of different environments. This thesis presents a realization for software systems of which only code is available. This code-based realization uses three measurement techniques for making necessary estimations: change analysis, fault-proneness models of software structural quality, and test coverage analysis.

The performance of the code-based realization is empirically evaluated as a prioritization technique through three sets of experiments. The first experiment, concerning the effects of involved parameters, indicates that the fault detection abilities is seldom affected by the involved parameters but is often improved by the use of feedback, specially on objects with TSL test suites. The execution time of the implemented techniques has been found to be impacted by the involved parameters intensively. The second experiment, comparing the code-based realization against those of the literature, reveals that on some objects BN-based techniques achieve a better rater of fault detection and on others perform comparatively. The third experiment, addressing same questions only with a different fault type, suggests that even when mutation faults do not reveal any significant difference, the performances of BN-based techniques can be higher if real (or more representative) faults exist in the objects. The presented experiments clearly demonstrate the advantages of the proposed framework.

To conclude, this dissertation has presented a new line of research in software regression testing. The published results from this research work (*e.g.,* [37, 93, 94, 96]) has been promising. Future research can be conducted to expand this work.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 402–411, May 2005. 59

[2] T. Apiwattanapong, A. Orso, and M.J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 2–13, Sept. 2004. 17

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996. 20

[4] Saïda Benlarbi and Walcelio L. Melo. Polymorphism measures for early risk prediction. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 334–344, 1999. 20

[5] James M. Bieman, Sudipto Ghosh, and Roger T. Alexander. A technique for mutation of java objects. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, page 337, 2001. 59

[6] David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997. 9

[7] Shawn Bohner and Robert Arnold. An introduction to software change impact analysis. *Software Change Impact Analysis*, pages 1–26, 1996. 18

[8] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996. 1, 18

[9] L.C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 252–261, 2002. 9, 11

[10] Lionel Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 56:98–167, 2002. 20, 32, 65

[11] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-orientedsystems. *Empirical Softw. Engg.*, 3(1):65–117, 1998. 20

[12] Lionel C. Briand, John W. Daly, and Jurgen K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999. 20

[13] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng.*, 28(7):706–720, 2002. 20, 21

[14] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and validating measures for object-based high-level design. *Software Engineering*, 25(5):722–743, 1999. 20

[15] P. A. Brown and D. Hoffman. The application of module regression testing at TRIUMF. *Nuclear Instruments and Methods in Physics Research A*, 293:377–381, August 1990. 9

[16] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *Proceeding of the Workshop on Domain specific approaches to software test automation (DOSTA)*, pages 1–7, 2007. 14, 16

[17] Timothy Alan Budd. *Mutation analysis of program test data.* PhD thesis, Yale University, 1980. 59

[18] Mei-Huei Tang; Ming-Hung Kao; Mei-Hwa Chen. An empirical study on object-oriented metrics. In *Proceedings on International Symposium on Software Metrics (METRICS)*, pages 242–249, 1999. 20

[19] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduction using extended dependence analysis. In *Fourth International Workshop On Software Quality Assurance (SOQUA)*, pages 62–69, 2007. 11

[20] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 211–220, 1994. 9, 10, 11

[21] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 197–211, 1991. 20, 45, 53, 65

[22] Michael Stepp Christian Collberg, Ginger Myles. An empirical study of java bytecode programs. Technical Report TR04-11, Department of Computer Science, Univeristy of Arizona, 2004. 45

[23] M.-C. Chung, C.-M.; Lee. Inheritance-based object-oriented software metrics. In *Proceedings on IEEE International Conference on Technology Enabling Tomorrow : Computers, Communications and Automation towards the 21st Century (TENCON)*, 11-13 Nov 1992. 20

[24] CKJM, 2008. http://www.spinellis.gr/sw/ckjm/. 53

[25] Jacob Cohen, Patricia Cohen, Stephen G. West, Leona S. Aiken, and Anonymous. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, volume 3. Lawrence Erlbaum Associates, 1994. 37

[26] Christian S. Collberg, Ginger Myles, and Andrew Huntwork. Sandmark–a tool for software protection research. *IEEE Security & Privacy*, 1(4):40–49, 2003. 18

[27] J. S. Collofello and J. J. Buck. Software quality assurance for maintenance. *IEEE Softw.*, 4(5):46–51, 1987. 17

[28] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks (research note). *Artificial Intelligence*, 42(2-3):393–405, 1990. 25

[29] Robert Cowell. Introduction to inference for bayesian networks. In *Proceedings of the NATO Advanced Study Institute on Learning in graphical models*, pages 9–26, 1998. 25

[30] Philip B. Crosby. *Quality Is Still Free.* McGraw-Hill, 1996. 19

[31] P. Dagum and R.M. Chavez. Approximating probabilistic inference in bayesian belief networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):246–255, 1993. 25

[32] Giovanni Denaro, Sandro Morasca, and Mauro Pezzè. Deriving models of software fault-proneness. In *Proceedings of the IEEE International Conference on Software Engineering and Knowledge Engineering (Seke)*, pages 361–368. ACM, 2002. 20

[33] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 339–348, 2001. 15

[34] http://www.gnu.org/software/diffutils/, 2008. 17

[35] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, 2006. 59, 67, 78

[36] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005. 57, 59

[37] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, page to appear, October 2008. 62, 99

[38] Hyunsook Do and Gregg Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 141–151, 2006. 9, 11, 57, 62

[39] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering: An International Journal*, 11(1):33–70, 2006. 13, 60, 78

[40] T. Dogsa and I. Rozman. Camote-computer aided module testing and design environment. *Software Maintenance, 1988., Proceedings of the Conference on*, pages 404–408, Oct 1988. 9

[41] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 170–179, 2001. 22, 46

[42] Sebastian Elbaum, David Gable, and Gregg Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the IEEE International Symposium on Software Metrics(METRICS)*, pages 169–179, 2001. 21

[43] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Softw. Eng.*, 28(2):159–182, 2002. 13, 16, 43, 60

[44] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3), 2004. 13, 16

[45] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56(1):63–75, 2001. 20

[46] Emma, 2008. http://emma.sourceforge.net/. 53

[47] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000. 1

[48] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, Ltd., 1991. 16, 19, 29

[49] K.F. Fischer, F. Raji, and A. Chruscicki. a methodology for retesting modified software. In *Proceerdings of National Telecommunication Conference*, pages B6.3.1– B6.3.6, 1981. 9, 10

[50] International Organization for Standardization. *ISO/IEC 12207:1995 Information technology – Software life cycle processes*, 1995-2008. 1

[51] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., 1978. 1

[52] Genie/Smile, 2005-2008. http://genie.sis.pitt.edu/. 39, 53

[53] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Not.*, 10(6):493–510, 1975. 21

[54] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001. 11

[55] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005. 20, 21, 65, 66

[56] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., 1977. 20

[57] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, 1977. 59

[58] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Meth.*, 2(3):270–285, July 1993. 9

[59] M.J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2001. 9, 10

[60] M.J. Harrold and M.L. Souffa. An incremental approach to unit testing during maintenance. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 362–367, 24-27 Oct 1988. 9, 10, 18

[61] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 07(1):31–36, 1990. 9, 10

[62] David Heckerman. A tutorial on learning with bayesian networks. *Learning in Graphical Models*, pages 301–354, 1999. 25

[63] Jason Osborne Hema Srikanth, Laurie Williams. System test case prioritization of new and regression test cases. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 64–73, 2005. 14, 16

[64] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity.* Prentice-Hall, Inc., 1996. 20

[65] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981. 20

[66] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. *SIGPLAN Notices*, 25(6):234–245, 1990. 17

[67] IEEE – The Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1219-1998: IEEE Standard for Software Maintenance*, 1998. 1

[68] D. Jackson and D.A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. *Software Maintenance, 1994. Proceedings., International Conference on*, pages 243–252, Sep 1994. 17

[69] Dennis Jeffrey and Neelam Gupta. Test case prioritization using relevant slices. In *Proceedings of International Computer Software Applications Conference*, pages 411–420, September 2006. 13

[70] Finn V. Jensen. *Bayesian Networks and Decision Graphs.* Springer, 2001. 25, 51

[71] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 92–, 2001. 13

[72] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., 2002. 19

[73] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 119–129, 2002. 14, 35, 55

[74] Ross Kindermann and J. Laurie Snell. *Markov Random Fields and Their Applications.* AMS, 1980. 23

[75] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analysis for Java software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska–Lincoln, April 2006. 53

[76] Kevin Korb and Ann E. Nicholson. *Bayesian Artificial Intelligence*. CRC Press, Inc., 2003. 38

[77] B. Korel and J. Laski. Algorithmic software fault localization. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume ii, pages 246–252 vol.2, Jan 1991. 13

[78] David C. Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. On regression testing of object-oriented programs. *J. Syst. Softw.*, 32(1):21–40, 1996. 10, 18

[79] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, 1993. 22

[80] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282–290, Nov 1992. 18

[81] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Stat. Soc., Series B (Methodological)*, 50(2):157–224, 1988. 65

[82] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980. 1, 17

[83] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings on IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 442–453, 2003. 15, 16

[84] Hareton K. N. Leung and Lee J. White. Insights into regression testing. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 60–69, 1989. 2, 9

[85] H.K.N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 201–208, 1991. 11

[86] R. Lewis, D. W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *ESEC '89: Proceedings of the 2nd European Software Engineering Conference*, pages 487–496, 1989. 9

[87] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transaction on Software Engineering*, 33(4):225–237, April 2007. 14, 16, 57

[88] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., 1980. 1

[89] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc., 1994. 20

[90] Alexey G. Malishevsky, Gregg Rothermel, and Sebastian Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 204–213, 2002. 11

[91] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989. 20

[92] Tom Mens and Serge Demeyer. *Software Evolution.* Springer Publishing Company, Incorporated, 2008. 17

[93] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Bayesian networks. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE), LNCS 4422-0276*, pages 276–290, 2007. 99

[94] Siavash Mirarab, Afshar Ganjali, Ladan Tahvildari, Shimin Li, Weining Liu, and Mike Morrissey. A requirement-based software testing framework: An industrial practice. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, page to appear, September 2008. 97, 99

[95] Siavash Mirarab, Alaa Hassouna, and Ladan Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 177–188, June 2007. 18

[96] Siavash Mirarab and Ladan Tahvildari. An empirical study on Bayesian Network-based approach for test case prioritization. In *Proceedings on The IEEE International Conference on Software Testing Verification and Validation (ICST)*, pages 278–287, April 2008. 99

[97] T.M. Munson, J.C.; Khoshgoftaar. The detection of fault-prone programs. *Transactions on Software Engineering*, 18(5):423–433, May 1992. 20

[98] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing, Second Edition.* Wiley, June 2004. 2

[99] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S. Rosenblum. Using component metadata to regression test component-based software. *Software Testing, Verification, and Reliability*, 17(2):61–94, 2007. 11, 42

[100] T.J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6):676–688, 1988. 2, 58

[101] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988. 23, 39, 51

[102] Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. Status report on software measurement. *IEEE Softw.*, 14(2):33–43, 1997. 19

[103] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 287–301, 1993. 21

[104] R. E. Prather and Jr. J. P. Myers. The path prefix software testing strategy. *IEEE Trans. Softw. Eng.*, 13(7):761–766, 1987. 18

[105] Bo Qu, Changhai Nie, Baowen Xu, and Xiaofang Zhang. Test case prioritization for black box testing. In *Proceedings of the IEEE International Conference on Computer Software and Applications (COMPSAC)*, pages 465–474, 2007. 15

[106] Fred L. Ramsey and Daniel W. Schafer. *The Statistical Sleuth.* Duxbury Press, 1st edition, 1997. 69, 71

[107] David S. Rosenblum and Elaine J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, 1997. 22, 46, 62

[108] Gregg Rothermel and Mary J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, 1997. 9, 10

[109] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transaction on Software Engineering*, 22(8):529–551, 1996. 9, 10, 11

[110] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. Test case prioritization: An empirical study. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 179–188, 1999. 5, 12, 27

[111] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. on Softw. Eng.*, 27(10):929–948, 2001. 4, 5, 12, 13, 57, 60, 62

[112] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings on IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 281–292, 2003. 15, 16

[113] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings on The IEEE International Conference*

on Software Testing Verification and Validation (ICST), volume 0, pages 141–150, 2008. 14

[114] Sandmark, 2008. http://sandmark.cs.arizona.edu/. 18, 52

[115] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002. 13, 46

[116] A.-B. Taha, S.M. Thebaut, and S.-S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the IEEE International Conference on Computer Software and Applications (COMPSAC)*, pages 527–534, Sep 1989. 9

[117] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, page 44, 1998. 9, 10

[118] Kristen R. Walcott, Mary L. Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *ISSTA*, pages 1–12, 2006. 13

[119] Z. Wang, K. Pierce, and S. McFarling. Bmat – a binary matching tools for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2, March 2000. 17

[120] L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings on IEEE International Conference of Software Maintenance (ICSM)*, pages 262–271, 1992. 9

[121] Scott A. Whitmire. *Object Oriented Design Measurement*. John Wiley & Sons, Inc., 1997. 20

[122] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira A. Bellcore. A study of effective regression testing in practice. In *Proceedings on IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 264–274, 1997. 12, 60

[123] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991. 17

[124] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007. 9, 11

[125] C. Yuan and M.J. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *Conf. Uncertainty in Art. Intel.*, pages 624–637, 2003. 65

[126] Adam Zagorecki and Marek J. Druzdzel. An empirical study of probability elicitation under noisy-or assumption. In *Proceeding on The International FLAIRS Conference*, 2004. 26

[127] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *Proceedings of the IEEE International Conference on Quality Software (QSIC)*, pages 15–24, 2007. 13

[128] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. 21

[129] J. Ziegler, J.M. Grasso, and L.G. Burgermeister. An ada based real-time closed-loop integration and regression test tool. *Software Maintenance, 1989., Proceedings., Conference on*, pages 81–90, Oct 1989. 9

[130] Thomas Zimmermann, Peter Weigerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005. 18