A COMPONENT RANKING FRAMEWORK FOR MORE RELIABLE SOFTWARE

by

Dhyanesh Chaudhari

A thesis submitted to the Department of Electrical and Computer Engineering in conformity with the requirements for the degree of Masters of Applied Science

Queen’s University
Kingston, Ontario, Canada
(September, 2013)

Copyright ©Dhyanesh Chaudhari, 2013
Abstract

Software components are meant to be reusable and flexible by design. These characteristics and others continue attracting software developers to adapt a component (typically designed elsewhere) into their systems. However, software components are also vulnerable to reliability and security problems due to existence of non-obvious faults. We believe that a systematic approach to detect failures of a component and prioritize components using such failures can help developers decide on appropriate solutions to improve reliability. In this thesis, we present a framework that can help developers in detecting and ranking component failures systematically so that more reliable software can be achieved. Our proposed framework can allow monitoring critical components within a system under instrumentation, detecting failures based on specifications and using failure data and input from developers to rank the components. The proposed approach provides information for developers who could decide if the reliability could be improved by trivial code modification or require advanced reliability techniques. A prototype is designed along with a number of failure scenarios to detect specific failure types within a component. Four major failure types (value, timing, commission, and omission) are detected and used to rank software components. We conducted an experimental evaluation using two subject systems to assess the effectiveness of the proposed framework and to measure its performance overhead. Our experimental results show that the approach can benefit system developers by prioritizing components for effective maintenance with a minimal overhead.
Acknowledgements

I would like to thank my group members at Queen’s Reliable Software Technology group who provided me with their guidance, support and encouragement throughout my program.

My special thanks goes to Komminist Sisai Weldemariam for all valuable inputs and thoughts while writing this thesis.

Most importantly, I would like to express my heartfelt appreciation to my supervisor, Dr. Mohammad Zulkernine, for his patience and endless support. Without him, this wouldn’t be possible.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

(Dhyanesh Chaudhari)

(July, 2013)
# Table of Contents

Abstract ........................................................................................................................................... ii  
Acknowledgements ........................................................................................................................... iii  
Statement of Originality ................................................................................................................... iv  
Chapter 1 Introduction ...................................................................................................................... 1  
  1.1 Background ............................................................................................................................... 1  
  1.2 Motivation ................................................................................................................................. 2  
  1.3 Proposed Approach: Overview ............................................................................................... 4  
  1.4 Contributions ............................................................................................................................ 6  
  1.5 Organization of the Thesis ....................................................................................................... 7  
Chapter 2 Background and Related Work ...................................................................................... 8  
  2.1 Software Component and Development ............................................................................... 8  
  2.2 Improving Software Reliability ............................................................................................. 10  
    2.2.1 Terminologies ..................................................................................................................... 10  
    2.2.2 Overview of Reliability Techniques .................................................................................. 14  
  2.3 Related Work .......................................................................................................................... 18  
    2.3.1 Fault Detection ................................................................................................................... 18  
    2.3.2 Failure Classification ........................................................................................................ 21  
    2.3.3 Component Selection and Ranking .................................................................................. 21  
  2.4 Summary ................................................................................................................................. 26  
Chapter 3 Failure Detection and Ranking Framework .................................................................. 28  
  3.1 Running Example ..................................................................................................................... 28  
  3.2 The Proposed Identification and Ranking Framework .......................................................... 31  
    3.2.1 Design Assumptions for the Proposed Framework ....................................................... 32  
    3.2.2 Architecture of the Proposed Framework ...................................................................... 33  
    3.2.3 Operation of the Framework .......................................................................................... 39  
  3.3 Failure Classification and Scenarios ..................................................................................... 42  
    3.3.1 Timing Failure Scenarios ................................................................................................. 43  
    3.3.2 Value Failure Scenarios .................................................................................................... 46  
    3.3.3 Commission Failure Scenarios ....................................................................................... 49  
    3.3.4 Omission Failure Scenarios ............................................................................................. 51  
  3.4 Ranking Algorithm ................................................................................................................. 53  
    3.4.1 Weighted QoS .................................................................................................................. 54
List of Figures

Figure 2.1: A simple example of component based software system, namely a hypothetical text processing software................................................................................................................................. 9
Figure 3.1: A simplified view of the Shopping Cart system......................................................................................................................... 29
Figure 3.2: A simplified view of the proposed detection and ranking framework................................................. 34
Figure 3.3: Snippet of code instrumentation example for monitoring execution time. ......................... 36
Figure 3.4: Example result as produced by the View component of the framework......................... 38
Figure 3.5: Phases of the proposed framework................................................................................................. 39
Figure 3.6: The operational workflow of the framework. ........................................................................ 42
Figure 3.7: Sequence diagrams for timing failure scenarios........................................................................ 45
Figure 3.8: Calculating QoSRank and arranging components in appropriate sets. ................................ 55
Figure 3.9: Calculating FailureRank......................................................................................................................... 56
Figure 3.10: Ranking each candidate component set based on the number of failures recorded. . 58
Figure 3.11: Overall ranking of the framework................................................................................................. 59
Figure 4.1: Sample output result showing ranked components for the Shopping Cart system. .... 68
List of Tables

Table 3.1: Examples of requirements and specifications for the Shopping Cart system .............. 30
Table 4.1: Summary of the target systems for our experimental evaluation. ............................. 63
Table 4.2: Failures detected and ranking order for an operational profile of the Shopping Cart system, where C1 = Delegate component, C2 = Processing component, C3 = Inventory component and C4 = Cart component. .................................................................................................................................... 67
Table 4.3: Percentage of failures detected and correct rank order for the target systems. .......... 69
Table 4.4: Change in component execution times for the Shopping Cart system. ..................... 72
Table 4.5: Change in component execution times for the CVS system .................................... 73
Table 4.6: Percent increase in execution time versus increase in number of APIs used for the target systems. ............................................................................................................................................ 75
Chapter 1

Introduction

1.1 Background

Various methods for developing software have been practiced with the primary objective of producing a software system that is capable of performing the intended service. The current trends in building a software show that the development practices are continuing on (re)using off-the-shelf components. One of the reasons is that of achieving maximum modularity and extensibility. There is also a growing complexity of applications and the constant time-to-market constraints. Thus, a well-engineered component can improve the reliability and security of a software system, reduce development time as well as save cost associated with the development process and maintenance [20, 21]. Alternatively, when components are reused, developers need to verify that these components perform their intended functions. This means that in order to ensure reliability, it is necessary to assure that there are no potential vulnerabilities in a component that contribute to a system to fail. Thus, critical components within a system must be identified and dealt with in a systematic manner. This thesis focuses on improving the reliability of a software system by identifying component failures of the system under analysis and prioritizing the faulty components based on such failure information.

Consider a shopping cart system that can be used by multiple web applications. If a component within the shopping cart system fails or deviates from its specifications, it can
lead to unpredictable behaviors including failures to provide the required services. For example, the “Checkout” component may fail to respond after a user has supplied his/her credit card information. Hence, in order to provide a highly reliable software system, techniques such as fault prevention, fault removal, fault tolerance, and fault forecasting are often used [3, 4, 5]. To prevent unpredictable behaviours and improve software reliability, there is a need for an approach to detect and notify failures to developers. If not detected, such failures could be silent or the underlying errors may propagate to other components and eventually could complicate maintenance.

1.2 Motivation
An ill-designed component can affect the reliability and security of the entire software system [47, 69, 70, 71]. A number of approaches have been proposed for reliability engineering to develop a highly reliable system [e.g., 1, 4, 36, 41, 48, 68]. These techniques can prevent, remove, or otherwise forecast faults within a software system at different development phases so that more reliable software can be achieved. Among other techniques, fault prevention and fault tolerance are well-established techniques for improving software reliability. Although complimentary to each other, fault prevention achieves the objective by eliminating faults (ideally all) while fault tolerance maintains the continuity of source in case of a failure [5]. These techniques could have been benefited from the results of a component ranking framework for effectively prioritizing components based on discovered failure types, before applying a particular mitigation technique under consideration.
Traditional component ranking frameworks use quality-of-service (QoS) properties [13, 14], service-user requirement [15], past component usage experiences, and software architecture and invocation links [16] to find faults in each critical component and prioritize them accordingly. Consequently, specific mitigation strategies can be selected and applied to classify failures effectively. However, without the domain knowledge of the component in a system or developer’s limited resources, it is difficult for these complimentary approaches to suggest an appropriate reliability technique.

We believe that components used within a system, and operated based on specific profiles have certain failure characteristics (fails in a evident way) when detected. Thus, failure types and ranking filters (set by a developer) can lead to a more effective ranking technique that can help prioritize maintenance in the context of reliability. This adds pressure on software developers to create more reliable applications. Complimentary approaches for component ranking or selection may be ineffective if information such as cost, development resources, time, and other non-functional properties are not known. Application domain, operational profile, specifications, service delivered, and many other factors differ from software to software. Hence, quality of component ranking executed by complimentary approaches may be less practical. We believe that software reliability techniques such as fault removal, fault prevention, or fault tolerance could be more effective if failure characteristics of a software component are known. Given the limited resources provided to software developers, aforementioned techniques can be expensive.
Failure to mitigate problems without knowing failure characteristics of a component can result in a waste of developer’s crucial resources. We envisage that component ranking and selection, when failure types are provided, can be more constructive in mitigating failures for improved software reliability. The effectiveness of maintenance could be greatly enhanced if a resource prioritization is done. Such prioritization can allow to select the “top-k” failing components for a better trade-off strategy than trying to mitigate all failing components in a limited time at once.

1.3 Proposed Approach: Overview

Our specific focus include proposing a systematic approach to detect failures independent of any particular specification language, implementing scenarios and the algorithm that classifies an event scenario as a failure, and localizing events that cause failures and the possible events that propagate the error.

Existing techniques used for making software more reliable are very subjective to the target applications [21]. Detailed specifications, software requirements, service properties, and other artifacts are required to properly address the failure issues and mitigate failures using an appropriate reliability technique. The proposed framework, when hooked in a target system using the provided APIs, can instrument code in a component under monitoring, detect and classify failures into specific types. The recorded data, obtained during component monitoring, is used to identify failures and rank components.
There are three main phases of the proposed framework. In the first phase, the framework is integrated with the software under test to monitor candidate components. The application developer uses APIs of the framework to hook candidate components for monitoring. The developer identifies critical points and uses appropriate monitoring functions. During the software execution, when the control reaches at a critical point, the component sends the necessary runtime data along with event specifications required by the component to the framework. In the second phase, the proposed framework first identifies the API called and then analyzes runtime data and specification to detect and classify failures. If applicable, failure data along with other necessary attributes are stored in a database. During the third phase, at the end of software execution, using the available failure data and input filters from the developer, candidate components (i.e., components that are being monitored for failures by the framework) are ranked from the most failing to the least. Input filters are parameters supplied by the developer to enhance or refine the ranking process. Using the ranking results, the developer can decide on the reliability or mitigation techniques that will be appropriate for the faulty components.

We evaluated the effectiveness and performance overhead of the proposed framework by taking two real-world examples: a relatively small-scale custom e-commerce system (i.e., the Shopping Cart System) and an open-source version controlling system (i.e., CVS). Our experimental evaluation shows that the approach is able to detect and classify failures based on general scenarios and interfaces used by the developer with a minimal performance overhead.
1.4 Contributions

The proposed approach uses the detected failure data and general scenarios to classify failures and rank the participating components. The primary challenges to develop this framework are (i) Generalized approach to detect failures without using any particular software specification language; (ii) Implementing scenarios and the algorithm that classifies an event scenario as a failure; and (iii) Most importantly, defining criteria for ranking components to provide a systematic way to help improve software reliability. We addressed these challenges by contributing in the following way:

- The failures are detected and classified into four major categories: value failure, timing failure, commission failure, and omission failure. A set of scenarios are implemented and cataloged into the four failure types.
- Developed application-programming interfaces (APIs) for the framework that can be used by developers to monitor an event or a function call within a component. These APIs are developed in the same programming language (currently, only C/C++) as the system, so that the developers require no additional knowledge.
- Developed a monitor within a framework that hooks into the system and utilizes the expected and actual monitoring parameters to notify developers.
- Developed a ranking algorithm that uses classified failure data, input parameters from the system developer, and other criteria to output customized component ranking for the target system.
1.5 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we provide background discussion on component-based software development and other necessary software reliability terminologies and techniques to better understand the proposed approach. We also discuss related work with respect to component ranking and mitigation techniques. Chapter 3 presents the proposed approach. The chapter also discusses the implementation details of the prototype framework, assumptions made, as well as scenarios and the algorithm that classifies failures within a component. The component-ranking algorithm is also introduced and described. Chapter 4 presents the experimental evaluation of the proposed framework using two case studies. Finally, Chapter 5 draws conclusions and discusses the limitations of this work and possible future work.
Chapter 2

Background and Related Work

This chapter provides an overview of the background materials and the related work. More specifically, Section 2.1 defines component and component-based software system. Section 2.2 discusses software reliability. It provides an overview on software faults, errors, and failures along with the techniques that can improve software reliability. Section 2.3 discusses the related work that focuses on complimentary approaches for component ranking and software reliability.

2.1 Software Component and Development

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.” [20]

In general, components within a system are designed to take specific responsibility by communicating with other components to comprise the system. Components provide several benefits such as cost-reduction, reusability, maintainability and ease of assembly [21]. Although very subjective, a component can be classes, files, packages, or modules that provide contractual services to the software using defined interfaces. For the purpose of this thesis, the term “component” could be any such entities as defined by the software developer. In our case studies, class files that provide services to the software system are
defined as components. A component is developed with contractually specified interfaces that can facilitate the integration with other components and assure maintainability. Since a component is an independent unit within a software system, it can easily be modified or replaced according to a change in software requirements. Notice we assume that a component should be tested prior to integrate it within a system with substantial confidence and minimal development (or maintenance) effort.

The development process of a system might be different from that of a component. Development of a component-based software system is carried out based on the assumption that the components have already been developed and used in other systems [73]. However, the development process of a component assumes the possibility of its reuse in various systems. Component-based system development may use the traditional software development model (e.g., Waterfall model [74]) where the implementation phase is replaced with component integration phase.

![Figure 2.1: A simple example of component based software system, namely a hypothetical text processing software.](image)

9
Figure 2.1 shows a simple example of a component-based software system. In the figure, a hypothetical text processing software with several components are shown that are linked with specified interfaces. Components such as “Spell-Checker” can be reusable in other systems. Each component in the figure has a specific goal and together constitutes a modular architecture for the system. In this thesis, we assume that a component-based software system will contain independently performing components.

2.2 Improving Software Reliability

In this section, we provide some terminologies used throughout this thesis and provide an overview of software reliability techniques.

2.2.1 Terminologies

Software dependability designates some properties and allows us to place assurance in a system. It includes other software quality attributes such as reliability, availability, safety and security [25]. We focus on software reliability aspect, specifically, how the software behaves and how often it deviates from the specification causing noticeable failures. As we will use terms such as faults, errors, and failures throughout the thesis, we define these terminologies here.

“Fault is a physical defect, imperfection or flaw that occurs in hardware or software.” [26]

In software systems, common faults could be architectural, incompatible interfaces between components, runtime environment, bug in the system, etc. Software faults can result in errors.
“Error is a deviation from correctness or accuracy.” [26]

Errors are effect of faults in a software system. Any deviation from the specified requirements, including correctness of output values, or results generated from the system (or component) is considered as a software error. An error produced in a system can lead to one or more failure.

“Failure is a non-performance of some actions that is either due or expected.” [26]

Software failure is a deviation from the service that the system is supposed to deliver. Some software systems lack well-defined specifications. This may cause to ignore some failures caused by the system. Failure is caused due to error. However, not all errors cause failures. Errors produced in a system may lead to circumstances where the system can still perform to specifications (i.e., when the faults are masked). For example, a bug in a software component is a fault. When the software is executed, this bug may lead to an error. If the error is not handled by the component, it may cause the component to fail.

In this thesis, we focus on four major failure types: value failure, timing failure, commission failure and omission failure. We will define these in the context of component-based systems. Value failure occurs when an incorrect value subsists, returned by a component or any of its function calls. Timing failure occurs when a component fails to deliver the service within specified time constraints. Time specification could be absolute or could be relative to start or end of other events within a component or a system. Commission failure occurs when a service is delivered or an
operation is executed and results are returned unexpectedly. Such unexpected execution and delivery could be intrusive to the system. Omission failure occurs when a component or an operation does not return a service or value when expected. All these failure types can lead to unexpected behaviors within a component or a system, ultimately challenging the reliability of the software.

“Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment” [27].

A number of definitions about software reliability can be found in the literature, which often states reliability in qualitative or quantitative (in terms of failure probability) manner. To achieve certain reliability, it may be required to test operational profiles, measure and localize faults, and calculate reliability at every step of the development cycle (since specific time and requirements are part of the reliability equation). In order to improve reliability and achieve a set of reliability goal, techniques such as fault prevention, fault removal, fault forecasting or fault tolerance may be necessary [3, 4, 28]. Nonetheless, these techniques require certain prerequisite data from the target system. To effectively use any of these techniques, data such as fault types, location, time of failure and operational-profile under execution is necessary [27]. In a component-based software system, due to its modular architecture, a fault must be localized at a component level where the above techniques could be used to improve reliability.

We argue that the effectiveness of the techniques can substantially be improved if the fault can be localized to a component level and if they can be classified into specific
types. The proposed framework focuses on component-based software systems where the
data collected could be used to effectively rank component (in terms of failures) and
assist in choosing reliability improvement techniques. The ranking algorithm of the
proposed framework identifies target software components as candidate and comparable
components.

A candidate component is a software component that is being monitored
for failures by the framework.

Candidate components are the components that the developer chooses to hook into the
framework using monitoring APIs.

A comparable component is a functionally equivalent component that is
available to the developer and can be exchanged with a candidate
component without any loss of acceptable services.

A comparable component could be used for fault tolerance or to compare functionally
equivalent components. The ranking algorithm of the proposed framework distinguishes
monitored components into three sets – Priority, Secondary and Other set.

Priority set is a set of candidate components that require primary
attention to help improve software reliability.

Secondary set is a set of candidate components that have a better
comparable component available to the developer or has an acceptable
failure rate.
Other set is a set of components that are only ranked using the failure data and assigned failure types.

Components in Other set will not participate in ranking based on QoS values or failure rate values. The Priority, Secondary and Other sets are mutually exclusive.

2.2.2 Overview of Reliability Techniques

In this section, we discuss four major techniques that could be used to improve software reliability. These techniques are fault prevention, fault removal, fault forecasting, and fault tolerance. Due to the intrinsic complexity of software systems, it is difficult for a software developer to depend on any single technique to improve reliability. Depending on reliability goal and software requirements, a combination of any of these methods can be used. There exist many other methods as discussed in [9].

**Fault prevention** – “Technique to prevent fault occurrence or introduction [28].”

This technique emphasizes on improving software reliability by applying structured and methodological process that avoids fault occurrences. Specific guidelines, software specifications, and many other methodological approaches throughout software development cycle help prevent software faults from being introduced [28]. Component-based software development provides another approach that, through component reuse, offers a measure to minimize faults within a system. It is based on the fact that components previously used are tested and applied in a working system for delivering specific services. Software maintenance can assure that a component executes with
respect to specified requirements and can be reused. Existing fault prevention techniques can be applied during specification, design, development, or deployment phase of a software development cycle. Prevention during the requirement phase could include formalizing software specifications to remove any ambiguity. Prevention during the design phase takes into account the adoption of specific procedures and choice of tools [29]. Component reuse, object-oriented paradigm, automated tools, etc. is used to prevent faults within a system.

Activities such as rigorous testing of a system (or a component), online or offline software monitoring, fault removal with code modifications falls under fault removal technique.

**Fault removal** – “Technique to detect, by verification and validation, the existence of faults and eliminate them.” [28]

Building test cases, executing software system under the required environment, comparing actual and expected result data, and other methodological approaches are effectively applied to localize and remove faults within a system. Software testers can be employed to help detect and eliminate faults with the goal of improving software reliability prior to deployment [30].

**Fault forecasting** – “Technique to estimate, by evaluation, the presence of faults and the occurrence and consequences of failures.” [28]

Fault forecasting predicts the likelihood of faults so that they can be eliminated to improve system reliability. Fault removal together with forecasting is the way to reach
confidence in the software to deliver the required and specified services. Faults can be forecasted using reliability estimation as well as prediction. Failure data obtained during testing and operational phases of the system can be used to determine quantitative reliability of a system. Using past failure data available through testing or execution under operation profiles, software domain knowledge and error propagation data can be used to predict the reliability of a system. The existing fault forecasting techniques can reveal whether additional steps are necessary to improve reliability. These techniques do not reveal the incompleteness of the requirements of system under investigation [3]. Fault forecasting with a reliability goal in mind may indicate the need for alternative techniques such as fault tolerance.

**Fault tolerance** – “Technique to ensure a service capable of fulfilling the system’s function in presence of faults” [28]

Fault tolerance is a mechanism of masking faults and failures and delivering desired service stated under specifications. Fault tolerance allows achieving the intended reliability by acquiring the most reliable functionally equivalent components with given cost, time and performance constraints. There are situations where fault removal or fault prevention cannot be applied. For example, off-the-shelf components cannot be modified, while for some components, the maximum reliability measure may not be achievable. In these situations, fault tolerance techniques attempt to ensure a greater degree of system reliability. Mostly, fault tolerant systems are designed to deliver full services under occurrence of any faults while in few cases the system is designed to provide limited
services until the fault is removed. These systems are known to have “fail-soft” tolerance capability [32].

Typically, fault tolerance can be applied by providing redundant components, or functionally similar components [36]. Multiple instances of similar components are applied to a system for fault tolerance. A primary component is executed first, and upon failure, other instances are activated. Multiple instances of functionally similar components, developed independent of each other are used in a series or in parallel to cope with faults in a system. Recovery Blocks (RB) [19], N-Version Programming [18], Retry Blocks (RtB), and N-Copy Programming are some classic and basic techniques used for fault tolerance. Decision mechanisms such as voting algorithms, acceptance checks, and reasonableness tests can be used to ensure the correctness of results generated within a component [36]. These techniques help stabilize a system under faults and allow them to move forward. However, these are limited to cope with anticipated faults and cannot guarantee to mask all possible fault types (known and unknown).

Extensive research has been done on developing reliability techniques to improve the overall software reliability. However, the techniques can only be effective when a system developer analyzes and identifies failure characteristics within the software (or component). In order to accomplish this goal in an effective manner, faults must be detected, failures must be classified, and essential components must be selected for maintenance. The next section discusses some related work on component-based reliability analysis.
2.3 Related Work

A number of approaches and techniques to improve software reliability have been proposed in the past. We discuss here how our work relates with some of these existing component-based reliability analysis approaches: (1) fault detection, (2) failure classification, and (3) component ranking and selection.

2.3.1 Fault Detection

An extensive research progress has been made in the area of fault detection. Network-based services and distributed components running on different remote nodes use keep-alive messages or SNMP [35] to identify failures within a network layer. Approaches such as [36-39], with the help of middleware, can identify failures in the network layer but fail to address failures within an application.

Jass [53] is a design-by-contract extension that allows specifications in the form of assertions in Java language. Jass pre-compiler translates these annotations into Java code and the compliance with specifications is tested during runtime. These assertions are boolean expressions with specific keywords. Design-by-contract, as proposed by Meyer [54], is applied here as it allows assertions as pre/post conditions, loops, class invariants, and check statements. The Temporal Rover [55] is a specification-based verification tool that combines formal specification (Liner-Time Temporal Logic and Metric Temporal Logic), with execution-based testing. The monitoring application can be applied to systems written in C, C++, Java, VHDL, or Verilog. Specifications written as comments
within the source code is converted into executable code by the parser, and compiled and linked as part of an application.

Similar to Jass [53], the proposed framework has automated response and is used for general purpose monitoring of a system. It can also be used to target embedded and concurrent systems. Our approach, when compared to Jass and Temporal Rover, does not have any response-effect within an application under monitoring. It is a non-intrusive monitoring that could be used to detect faults and check if a fault within a component can propagate to other components within an application. In Jass, response to violation could be automated with exception handling. Instead of assertions that require a specific pre-compiler (or parser), our approach uses native programming language (C++) to specify requirements as expected values. This reduces the learning curve involved to integrate the monitoring framework and apply target software specifications into the framework. It also eliminates the use of any customized pre-processor or compiler to be used for software implementation. This aids in saving limited time and budget resources to improve software quality through reliability.

Other production or research level monitoring tools (e.g., [56-60]), although more advanced, require more resources to interpret software specifications and integrate it within a target system. Some tools suggest automated dynamic or static monitoring points, while others differ in the software abstraction level. This level of monitoring may not be necessary as an alternate approach of implementing language specific APIs to monitor and log data could very well be sufficient.
Apart from the previous approaches to dynamic fault detection through monitoring middleware, fault prediction has also been the area of research. In [50], Ostrand and Weyuker mention that certain files within a system are more likely to contain identifiable faults. The mentioned model uses previous fault data and statistics, and forecasts files that are likely to contain large records of faults. In this approach, the tester plays an important role in fault detection. The approach is similar to ours in a way that we allow developer to be a key entity in choosing subsystems that can contain most faults. However, we do not use any statistical model and past fault records to predict such files. We rely on current test set and observable data to detect and localize faults within a system. Hassan and Holt in [51] describe a heuristics method to provide top-ten list of vulnerable subsystems where the fault can exist. Project managers can use this list to focus their testing resources (human as well as software).

These techniques (discussed in [50, 51]) differ in their approach but work towards a similar goal of achieving a fault free system. Fault detection, prediction and forecasting models can reduce the testing duration and failure risks involved during the software delivery (or deployment). To remain competitive in the fast paced software development world, it becomes absolutely necessary to prioritize your limited development resources. Hence, our approach (that uses pre-existing monitoring approach) to detect faults and characterize components (based on failure) could be more beneficial.
2.3.2 Failure Classification

Mozilla Firefox, Google Chrome, and Microsoft Windows introduced special modules that detect and report (with user’s permission) failures systematically for maintenance. Such automated support collect all the diagnostics information such as program stack, state, and input that could be used to classify failures into certain category. Techniques such as cluster analysis and multivariate visualization are used to group failures into automated categories. These clusters are based on failure similarity. Failures are then prioritized based on frequency and rate of occurrence.

Some techniques [e.g., 41-45] are limited to grouping similar failures in a cluster and visualizing any module that need immediate attention. Our approach to failure classification involves instrumenting the software under monitor and collecting data. The composed data is then analyzed for expected executions based on specifications provided by a developer. These target software specifications are provided as input to the application interfaces. Based on the scenarios selected during instrumentation phase, the failure is classified into one of four categories. The classification of failures is according to [72], with the aim of treating software failures while considering underlying failure types.

2.3.3 Component Selection and Ranking

Component selection and ranking have become an essential approach to choose where your development resources (human and software) should be managed. In case of applying fault tolerance strategies in component-based systems, prioritizing and
localizing fault tolerance (or other reliability techniques) to specific component can increase the overall system reliability with relatively small amount of cost and performance overhead. Component selection and ranking provide a reasonable trade-off for improving software reliability. Techniques in component-based reliability prediction can be localized to specific components that has higher failure rate and are prone to specific failure types [46, 47, 48]. Existing component selection and ranking approaches can be categorized into three types: architecture-based, QoS-based (Quality of Service), and failure probability-based.

The authors in [13] proposed a QoS-aware middleware for fault tolerant web services. The middleware is capable of dynamically adjusting optimal fault tolerance strategy based on QoS properties of available service replicas. The QoS properties are obtained by encouraging participants to share their individually acquired QoS data. The approach is specific for web-services where functionally equivalent replicas are readily available. The proposed model is used for optimal fault tolerance strategy selection. The dynamic fault tolerance selection algorithm does not consider other means of improving reliability of software components. The fault tolerance strategy employed for selection and ranking is expensive and requires readily available replicas, which contribute to added cost and other software development resources. The QoS property is beneficial, however, the assumption of getting values based on user participations is challenging. In this case, our ranking algorithm uses failure data to rank components when fault tolerance strategy is not feasible. Although QoS is beneficial, situation may exist where an application may
not have any available replicas or pre-existing QoS values. Moreover, system analysts do not always want to record QoS information for the components. Challenge arises when software analysts are not willing to share such information. If enough participants are not present, the acquired data could lead to an inaccurate strategy selection.

Our approach does not limit to any specific application or assumes (or absolutely depend on) that QoS values are readily available or mandatory. In our proposed framework, the ranking of a component based on QoS information is optional and could be integrated if the developer has access to functionally similar replicas with acceptable QoS values. Integrating QoS and failure rate values (if available) could enhance the ranking process and provide better ranking insights. The proposed framework could be integrated with an application using provided interfaces and components could be ranked only using test-cases and operational profile scenarios.

Another approach in [15] uses a framework where component failures in web services are handled by fault tolerance strategies. A strategy selection algorithm employs QoS values and service-user requirements for selecting optimal fault tolerance strategy. Their proposed framework can be used to observe QoS performance of web-services at different locations (under different runtime environment) and store historical evaluation results. This employs an easy way to obtain data from the participating users. However, the algorithm (and framework) does not consider active faults and failures occurred within the system. Client-side testing of system components, however infeasible, can be more effective to derive practical results.
CloudRank [14] is QoS driven framework to rank components for cloud applications. The framework takes advantage of past component usage experience from different component users (e.g., application designers). This is specific to cloud applications and adopts very strict assumptions. The approach uses properties from different component users and applies the data to rank and select components. Usage experience may vary depending on the software execution environment and requirements, and may not be applied to any other system that differs in runtime environment or specifications. However, the approach convinces that ranking components provides a systematic approach to prioritize development resources in improving software reliability. Therefore, the proposed framework in this thesis takes inspiration from these kinds of work to systematically improve software reliability in non-cloud setting.

FTCloud [16] identifies component in software application based on architecture and invocation links in and out from different components. The approach is inspired by PageRank algorithm as used by Google search engine [49]. The approach has three phases: component graph building, component ranking and significant component determination. In component graph building, a Cloud application is architecturally modeled as a weighted graph where a node represents a component and directed edge between nodes represents component invocation relationship. FTCloud does not rank components based on relevant operational profiles or test cases. Also, it does not consider failure characteristics and specifications of a component that could be used to apply more effective reliability techniques.
More specifically, FTCloud focuses on two major issues (see [16]) – component selection algorithm and component ranking strategy. Their selection algorithm (optimal fault tolerance strategy selection) is based on failure probability, cost and response time. The component ranking is based on architecture dependencies and invocation links. The approach could work for a system where a developer does not have to consider additional non-functional constraints and have the application candidates that could be used readily. However, it fails to consider the failure characteristics of a component. That is, different components serving different services fail differently. Each component is independent unit within a system that is assigned specific functions and failure characteristics are based on the roles the component executes. For example, a timer component in a system is more likely to fail on timing constraints than value-based constraints.

In our approach, components are not only ranked based on the number of failures but additional parameters that can be configured (as input parameters) to optimize the output of the ranking. For example, having weights on each failure-type could have effect on how the components are being ranked. In this example, a developer can put more weights on timing and value failures than commission or omission failures to prioritize maintenance if timing or value failures are more crucial to the system reliability. Hence, when timing and value constraints are ranked, the component will get a relatively higher priority. Both FTCloud [16] and our proposed approach are developed on the fact that by maintaining a small part of the most significant components, software reliability could be greatly improved.

25
Our proposed framework is complimentary to previous approaches. It allows the developer to select components that require the most attention to gain reliability. The obvious difference between our approach and the related work is that we rank and select the most failing components.

The importance of the component may not depend only on the architectural dependencies or invocation frequencies, but on operational profiles and failure data obtained during testing. According to [50], there are many other factors that must be involved in the selection process. Here, we only focus on component’s failure characteristics and the relevant failure data that, in addition to other factors such as deadline, cost, resource restrictions, available replicas, can be used to focus resources on improving reliability.

2.4 Summary

The chapter discussed background information about component-based software systems and reliability techniques that can be used to improve component-based systems. Key terminologies used within this thesis are also defined.

The chapter also presented related work in fault detection, failure classification, and component selection and ranking. The related work primarily emphasized on the research and development practices in component selection and ranking where QoS data acquired by developer participation, past component usage experience and architectural model are used to select and rank software components. We also observed that the existing complimentary approaches have strict assumptions on parameter selection and ranking
techniques, such as pre-defined QoS values, usage information on available replicas and software constraints that are relatively difficult to obtain. The related work on fault detection focus on monitoring applications that require formal specifications or additional knowledge of custom pre-processors or compilers to build a system. These related approaches require more resources that could very well be used to focus on identifying or prioritizing software maintenance.

In the next chapter, we propose a framework that monitors participating components within software using monitoring APIs specifically designed to detect four major failure types. A developer can integrate monitoring code within a component based on domain knowledge and component failure characteristics. At the end, the analyzed data is used to categorize failure and rank components based on failure types and can be optimized by the developer by integrating other ranking parameters.
Chapter 3
Failure Detection and Ranking Framework

In this chapter, we present the details of the proposed framework for the identification, classification of failures and ranking of components in a given software system based on the classified failures. We start by introducing a running example in Section 3.1, which will be used throughout the chapter for describing failure scenarios as well as for integrating the proposed framework into software under monitoring. Section 3.2 describes the framework architecture and some assumptions. While Section 3.3 details the failure scenarios, Section 3.4 discusses the implementation of the algorithm for analyzing the monitored failure data and ranking components. Finally, Section 3.5 summarizes the chapter.

3.1 Running Example

The running example used in this chapter is a Shopping Cart system implemented in C++. The system comprises of four major components. The system is primarily used to provide a better understanding of the framework and its underlying logic. Figure 3.1 provides an architectural overview and shows interactions of components within the Shopping Cart system.
Figure 3.1: A simplified view of the Shopping Cart system.

The four major components of the Shopping Cart system are: Inventory, Cart, Processing and Delegate. The Inventory component provides a list of items available to buy. It also keeps track of quantity, item ID, and other relevant data of available items. In addition, the Cart provides list of added items, gross total cost, and item descriptions. During checkout, the Processing component finalizes the cart, calculates tax for each item, and processes it for shipping with estimated arrival time. The Delegate component of the system allows user to view the inventory and items in the cart, and to process input commands from a user. It also delegates calls between the system’s components. Table 3.1 provides a summary of the key roles (and specifications) that each component within the system is responsible for. Possible failures due to underlying faults or specifications are also mentioned.
Table 3.1: Examples of requirements and specifications for the Shopping Cart system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Role and Specifications</th>
<th>Possible failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cart</td>
<td>Creates empty cart for the user.</td>
<td>Value, Commission, Omission</td>
</tr>
<tr>
<td></td>
<td>Add an item to the cart from Inventory, and checks availability. The cart cannot be finalized.</td>
<td>Value, Omission</td>
</tr>
<tr>
<td></td>
<td>Removes an item from the cart, and it must be a valid item in the cart. The cart cannot be finalized.</td>
<td>Value, Omission</td>
</tr>
<tr>
<td></td>
<td>Finalize cart so that it cannot be modified, and the cart is ready to be processed.</td>
<td>Value, Commission, Omission</td>
</tr>
<tr>
<td></td>
<td>Shows cart description: items, price, and total items.</td>
<td></td>
</tr>
<tr>
<td>Inventory</td>
<td>View inventory item description, price and quantity available.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gets an item from inventory. Takes 2 second to process.</td>
<td>Timing</td>
</tr>
<tr>
<td></td>
<td>View quantity for a specific item.</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>Increment or decrement quantity in an inventory.</td>
<td>Value</td>
</tr>
<tr>
<td>Processing</td>
<td>Process items within the cart. Takes less than 10 seconds per item.</td>
<td>Timing</td>
</tr>
<tr>
<td></td>
<td>Calculate tax for each item. Takes 2 seconds per item.</td>
<td>Value, Timing</td>
</tr>
<tr>
<td></td>
<td>Sets user information such as shipping address. It also uses credit card number for payment process.</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>Checkout cart. Provides estimated arrival date and total amount.</td>
<td>Timing, Commission</td>
</tr>
<tr>
<td>Delegate</td>
<td>Create cart per user instance.</td>
<td>Commission, Omission</td>
</tr>
<tr>
<td></td>
<td>Takes input commands and delegates input as function call to corresponding component.</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>Show software help/manual.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Controls other components and provides integration platform for component communication.</td>
<td>Timing, Value, Commission, Omission</td>
</tr>
<tr>
<td></td>
<td>Standard input and output. Validates user commands.</td>
<td>Value</td>
</tr>
</tbody>
</table>
In Table 3.1, possible failures for some roles and specifications are not mentioned as no possible failures are expected. The features mentioned in Table 3.1 will be used to create scenarios where a Shopping Cart component may generate faults leading to failure. For example, specifications for the *Processing* component requires cart to be processed in less than 10 seconds per item. Failure to do so could be considered as a timing failure within the component. If a component is being monitored for an expected or unexpected state change or transition, commission or omission failures could be a possibility. For example, when the cart is finalized for checkout (i.e., no more modifications are allowed), unexpected calls such as add or remove an item should be reflected as commission failure. More details on failure classifications, failure scenarios, and detection techniques will be described later.

### 3.2 The Proposed Identification and Ranking Framework

The scope of this work is mainly focused on failure classification and component ranking using failures identified during runtime. A system developer can input optional parameters such as QoS values and acceptable failure rate for optimizing component ranking. These optional parameters could further enhance the ranking to improve software reliability. Our approach requires defined software requirements and can be instrumented within a component to identify and classify component failures. The framework does not suggest any reliability techniques, but provides failure data and component ranking to the developer that can be used to prioritize components and
improve software reliability. We begin with the list of assumptions that allow us to focus on the criteria for ranking components.

### 3.2.1 Design Assumptions for the Proposed Framework

The proposed framework relies on some assumptions that must be fulfilled to successfully rank components. As mentioned earlier, the target software must be a component-based software system where each component is operated independently and linked with other components using specified contractual interface. Components must be clearly defined by a developer and the definition must remain consistent throughout the system. In our running example, a component consists of files (header and corresponding implementation files) that provide all the necessary interfaces (function calls) to fulfill the specified role. For complex systems, a component may contain multiple files.

A component within software could be made in-house or can be supplied by a third party developer. Whatever the case, the framework assumes that the source code is available, as it is required by a developer to insert APIs to monitor critical points. It assumes that no faults or errors in the system are caused by the incompatibility due to component integration. In case of our running example, the Shopping Cart system was developed without any third party components. Pre-complied third-party components within the system can only be integrated to monitor calls and return values from other components.

The proposed framework uses developer specifications to identify failures. Hence, it assumes that system requirements and specifications are clearly defined and critical
points for monitoring are identified by a system developer to provide expected results to the framework. To detect failures of critical (as defined by a system developer) state changes within a component, states, state-values, transitions and transition conditions must also be available. Since a component operates independently within software, the framework only detects state behavior within the components.

The framework provides monitoring APIs that a developer uses to hook the component (within a system) to monitor, allowing evaluating the source code by inserting appropriate monitoring code at the identified critical points. Overall, the framework is more appropriate to monitor components, detect failures, and rank components for the target system before its launch (off-site). Hence, reliability of the target system can be improved before it is deployed.

QoS and acceptable failure-rate based ranking features are optional in the ranking algorithm. These features could be used to further prioritize ranking of the components. For example, in the Shopping Cart system, if the developer has access to a replica that performs the same roles as an Inventory component, QoS values for the candidate component as well as available replicas can be included in the ranking process. Based on the number of failures and execution time of the component, the framework can detect actual failure rate for the candidate component.

3.2.2 Architecture of the Proposed Framework
To fulfill the purpose of identifying failures, then classifying based on scenarios and ranking components, the framework is modularized into five components. Each
component has a specific role and performs services independent of each other. The framework monitors software during runtime and stores data into a local repository (MySQL database). Since the monitoring is done during the software execution, the framework is designed to perform the specified tasks with negligible overhead. The integration and monitoring of software can be done with minimal developer resources (e.g., computing resources and development time).

![Diagram](image)

**Figure 3.2: A simplified view of the proposed detection and ranking framework.**

Figure 3.2 shows a simplified view of the proposed detection and ranking framework. *Monitor, Analyzer* and *View* modules use local repository for storing runtime data as well as other persistent storage data, such as input parameters from the developer, failure data,
Monitor controller persists information on candidate components and failure type detection on each candidate component. For consistent function calls across the components, the *Database Controller* regulates input and output to the storage. The developer uses the public APIs provided by the framework to integrate it within the software under monitoring where the output can be viewed post analysis.

**Monitor:**

It monitors and records data about the runtime behavior of the software under supervision. The data will be stored in a local repository for later analysis. For each monitored component, the developer uses the monitoring interfaces to provide specifications by identifying critical points—at which the interfaces are inserted—within a component of the software under analysis. During runtime, when the instruction reaches at each of the critical point, the instrumented software sends the actual runtime data (actual input) and expected results to the monitor. The *Monitor* records the runtime data from the target system into a local repository, which is then analyzed for failures. The interfaces are specific for each failure type. Since the developer has specifications at his/her disposal, he/she will define the attributes and behavior that needs to be monitored. The developer can use all provided interfaces to detect all failures or could be specific to any single failure type. For example, in Table 3.1, the Shopping Cart system requires timing and value failures to be detected in *Inventory* component. Hence, the developer can specify and limit the monitoring to value and timing based failures.
Figure 3.3 shows some of the critical points identified (and code modification) in the Processing component of the Shopping Cart system where a timing API is used to monitor the event execution time. Specification for this component is shown in Table 3.1.

```java
/* Processing Component */
...

void processCart(Cart* crt) {
    for (i=0; i < crt->numItems; i++) {
        int itemPrice = crt->getItemPrice(i);
        int taxed = calcTax(itemPrice);
        // process item
        ...
        // process complete for item
    }
}
```

```java
/* Processing Component */
...

// Measure exec time - must be less than (10*numItems) seconds
void processCart(Cart* crt) {
    // process begin
    Monitor->execTimeBegin(...);
    for (i=0; i < crt->numItems; i++) {
        int itemPrice = crt->getItemPrice(i);
        int taxed = calcTax(itemPrice);
        // process item
        ...
        // process complete for item
    }
    // process end
    Monitor->execTimeEnd(...);
}
```

Figure 3.3: Snippet of code instrumentation example for monitoring execution time.

As shown in the figure (Figure 3.3), one of the critical points identified is the `processCart(...)` event (class method) where the items within a cart must be processed for shipping within a specified time. Since this is a time-critical function (event), the system developer has inserted APIs to detect timing failure.
Function calls `execTimeBegin(...)` and `execTimeEnd(...)` notify the monitor about the start and end of the cart processing.

**Monitor Controller:**

Monitor controller keeps track of the input set by the target system developer and the information provided by the Monitor module regarding the software components that need to be monitored. The Monitor Controller module can be used to regulate monitoring at the critical points and monitor different failure types for different operational profiles or execution scenarios.

**Analyzer:**

The Analyzer module analyzes the data collected by the Monitor module. This module identifies an operational and failure scenario (based on [72]), and performs the analysis to classify the data for failure. When the Analyzer module classifies a failure, the failure information is saved in the repository that is then handled by the View module of the framework. For failures such as commission and omission failures, runtime data of other event calls are also analyzed to identify causal relationships. Post-runtime analysis certifies that data analysis (e.g., failure classification and result comparison) does not affect the execution time of software (or the component) under supervision.

**Database Controller:**

The Database Controller manages the read/write operations performed by the other modules on the local repository. This module also regulates all the I/O operations from/to the repository in order to ensure constant function call across all modules. The three
major modules of the framework (Monitor, Analyzer and View) use this controller. Reusable function calls make it easier to save and retrieve information from the repository. Each failure or monitoring data has a specific template that is managed by this module.

**View:**

The View module provides the results of the monitored and post-analyzed data in the formats requested by a developer. The ranking of each component based on failure types, cause-effect relationships between critical points, failure localization, and other data can be viewed using this module.

![Figure 3.4: Example result as produced by the View component of the framework.](image)
An example failure results in the Shopping Cart system for certain operational profile and a candidate component is shown in Figure 3.4. The example shows ranking of candidate components of the Shopping cart system and also shows how the framework’s View component displays the ranked results. The tabulated results on the top-right of the figure shows overall ranked results produced by the framework. The term “NA” indicates that the developer did not allow detecting of the particular failure type for the candidate component. Component-level results (bottom of Figure 3.4) can be requested to view identified failure types, event information and actual versus expected results for each component within the target system.

3.2.3 Operation of the Framework

The operation of the proposed framework has three phases (as shown in Figure 3.5):

Figure 3.5: Phases of the proposed framework

Phase 1: Code Instrumentation. The framework is integrated within the target software to monitor its participating components. A developer uses APIs of the framework to hook candidate components for monitoring, by identifying the critical points of the software under monitoring. During the execution, when the program
instruction reaches these critical points, the necessary runtime data along with event specifications are sent to the monitor.

**Phase 2: Failure Classification.** The framework identifies the APIs that are called and analyzes the runtime data and specification to detect and classify failures. Pre-defined failure scenarios for the four failure types are implemented within the framework. If a failure is detected, the failure data along with other necessary attributes are stored in a local repository. The repository is used for storing runtime monitoring data and other persistent storage data, such as input parameters from a developer, failure data, and event executions.

**Phase 3: Component Ranking.** At the end of software execution, available failure data and input filters from the developer are used to rank candidate components. The components are ranked from the most failing to the least.

The framework uses post-runtime analysis (analyzer) to compare data with the software specifications, and algorithm to classify failures and localizes the critical point within the component at which the failure occurred.

More specifically, the process begins with a system developer who wants to measure and classify failures within a software system, with the goal of improving the reliability of that particular component of the system or the system itself. The developer adds the framework as a library or source code to the system under monitoring. With clearly defined specifications, and application knowledge, critical points within each component of the system are identified. Using the monitoring APIs of the framework and defined
failure scenarios, the developer instruments the source code to add monitoring calls at the critical points.

For each candidate component under monitoring, the framework uses the monitoring function call (interface) along with the runtime data and specifications. For each data recorded by the monitor, the analyzer module verifies that if the data meets its specifications. If the specifications are not met, the failure data is classified into one of the four failure types based on monitoring call and specifications. The classification is obtained based on the interface used, specifications recorded, and scenarios generated at runtime. An appropriate scenario will be used to classify a failure. At this point, the identified failure is recorded into its corresponding category (represented in a table) that is used by the ranking algorithm along with external parameters set by a developer to rank a component under monitoring and output the ranked results.
Figure 3.6 shows the sequence diagram of the entire framework process showcasing the roles of the major framework components. It shows how software components are monitored and how the runtime data is collected and used to rank components. Monitor controller of the framework is not mentioned, as it is not actively involved with runtime data or component specifications. More details on the ranking algorithm will be provided in Section 3.4. However, it is necessary to understand how each failure is classified.

3.3 Failure Classification and Scenarios

Failure classification helps prioritize component mitigation (through ranking) and improve the overall software reliability. In what follows, we provide details on how each
failure type is detected based on the scenarios (implemented within the framework) occurring within a system. The framework does not provide any automated means for mitigating these failures but helps the process by providing runtime results and fault localization. The concept behind defining failures based on the component (or event) behavior and scenarios are explained in the following sections.

### 3.3.1 Timing Failure Scenarios

The effects of timing failure can be handled in different ways depending on the properties of a particular application. For time-sensitive services, faults generating failures could be removed by code modification (fault removal) or could be masked by designing the system to be fail-safe [64] (switching to fail-safe state when failure is detected) or time-elastic (handling occasional failures but expect certain coverage or probability of success). Since the techniques are subjected to developer’s resources, the framework is refrained from handling these mitigation issues.

The proposed framework can detect timing failure within a component according to the following event scenarios:

**Scenario 1.** When an operation is called **BEFORE** the specified time.

Usually, an operation will be a method or function call within a component that performs certain service. The system (or component) developer specifies the expected time for an event or operation to be called. Time specifications can be absolute or relative to other operations.

**Scenario 2.** When an operation is called **AFTER** the specified time.
This is similar to Scenario 1 with the only difference being that the framework detector will evaluate the scenario as failure if the specification restricts the component operation to be called after the required absolute or relative time. Both Scenario 1 and Scenario 2 are limited to operational call. The Monitor module does not monitor for any successful execution of an operation. These scenarios must be explicitly mentioned to the Monitor module. The next two scenarios handle such events.

**Scenario 3.** When an operation is called within a specified time but return before a specified time.

**Scenario 4.** When an operation is called within a specified time but returned after a specified time.

These scenarios occur when an operation is expected to return call or results within a specified time to ensure the execution of other time sensitive events. Failure to return within a specified time constraints could cause Scenario 3 or Scenario 4. Scenarios 1-4 can be combined to specify call or return of an operation within a range of specified (by the system developer) time limit. For example, if the monitoring interface for Scenario 1 and Scenario 4 is used together, a function call before a specified time and function return after specified time can be classified as a timing failure.

**Scenario 5.** There is no specified time to call an operation but the time to execute the operation exceeds the required limit.

Similar to the above scenarios, a developer has an option to specify absolute start and end times for an operation or he/she can provide explicit execution limit. The Monitor
module starts a timer at the start of the operation and ends when the operation returns. The **Analyzer** module examines the results and classifies as a timing failure on **Scenario 5** if the execution exceeds the specified limit. Figure 3.7 provides the sequence diagrams for some of the above scenarios using our running example.

![Figure 3.7: Sequence diagrams for timing failure scenarios.](image)

In the four scenarios depicted in Figure 3.7, the Checkout component of the Shopping Cart system of the running example calls `processOrder(…)` operation on the
Processing Component. All four scenarios use different specifications under which the operation executes.

For time-sensitive systems, a developer needs to know the cause and location of timing failure before applying any mitigation techniques. Mitigation can be as simple as a code modification, or may require a fault-tolerance technique on a component. Without any background information on failure type and cause, a randomly chosen reliability technique may be ineffective. While using the ranking algorithm, if a developer provides QoS properties and values for available replicas, the framework can provide results that suggest if the service quality of available replicas is better than the candidate component.

3.3.2 Value Failure Scenarios
To simplify value failure scenarios, we may assume that no other failure type or fault exists concurrently or prior to an event that could cause value failure. Using this assumption, we can create scenarios where value failure occurs. Values can be specific, within a range, a reference object, or contain state changes within a component. The following scenarios define the value failures detecting using our framework:

**Scenario 1.** When a component (as a whole), or an invariant, function call or variables within a component are not up to specifications.

The framework provides an interface to verify if the output of a component meets its specifications. For objects within a system, a developer is responsible for providing a function to compare such objects. If a system specification requires an invariant to be true (i.e., satisfying the specifications) at all critical points mentioned within a component, or
requires checking local variables within certain scope, or comparing return results of an operation, all such scenarios will be classified as value failure.

**Scenario 2.** When an operation fails to meet the specified precondition.

**Scenario 3.** When an operation meets pre-condition, but fails to meet post conditions.

If an operation requires state-transition checks, or state-value checks, or other complex scenarios, the developer must specify the data (specification data) to monitor using the specified interface. For example, a return result of True or False may only indicate if an operation finished its execution, but does not check if it performed all the expected results. During runtime, the Monitor will perform checks and record system’s runtime data into local repository, which will be analyzed by the Detector after runtime.

**Scenario 4.** When an unexpected state-transition occurs within a component.

In this thesis, we assume that state-based faults produce a subset of value failures. An incorrect transition to unexpected state within a component can lead to a value failure (transition value is incorrect). The developer specifies critical points within a component where a transition is expected. During runtime, the Monitor module keeps track of the current state and a list of possible transitions. If an unexpected transition occurs, the data is recorded in the repository. The framework will analyze data for expected and unexpected transitions, as it will classify any unexpected transitions as a value failure in
accordance with Scenario 5.

Scenario 5. When a valid state-transition occurs but provides incorrect state values.

The framework provides a medium to specify conditions for valid transitions as well as to check if the state contains correct values. If a valid transition occurs, the Monitor module verifies the conditions for correct state values as specified by the system developer. If an incorrect state value is found, the framework classifies the observation to lead to value failure using Scenario 6. For example, in the Shopping Cart system, a return value of the method call (in Inventory component) for getting an Item from the inventory can be a valid transition to certain state within a system. However, the value (an item) returned may be different, producing incorrect state value – leading to value failure. Scenario 5 and Scenario 6 assume that all previous state changes and values were up to specifications and no valid errors were transferred till the current point of detection. If the above assumption is not valid, then the detected failure might be caused by a previous failure and must be referred by the developer (in the View component).

Going back to our running example, let us create a scenario that involves three components interacting within a target system. Consider Inventory, Delegate and Cart components from the system. A user can view the inventory using the Delegate component. A user has an option to add items to the cart. The Delegate component calls functions within the Inventory that gets the item and adds it to user’s cart. A system specification requires that the total amount of items must remain constant. That is, the
total quantity in inventory before adding to the shopping cart must be equal to the quantity of items in the cart and the remaining quantity in the inventory added together. Hence, such invariant can be checked using the framework and failure to respect the requirements may lead to a value failure according to Scenario 2 (incorrect invariant value).

3.3.3 Commission Failure Scenarios

Commission failure inherits some scenarios from timing and value failures. Depending on the application domain, unexpected execution and delivery of services could be intrusive to the system. Commission and omission failures are a subset of value or timing failures but are considered as a special case as they are subjected to application specifications. Commission failures are classified into their own subclass due to the scenarios that characterize them. The following scenarios represent the scope of detecting commission failures using the framework:

*Scenario 1.* When an unexpected state change within a component interferes with the input or output values of other operations. This interference may lead to value failure besides commission failure.

If an unexpected operation or state change is triggered within a component, it may produce effects that could cause any of the attributes (e.g., variables or dynamic method calls) within a component to change. Such changes may lead to value based failures localized to other operations. Hence, the situation is classified as commission failure according to Scenario 1.
**Scenario 2.** When an unexpected operation or state change causes timing overhead on the system. If timing overhead is caused on a time-sensitive operation leading to any scenario in timing failure, it can be considered as commission failure.

This is similar to *Scenario 1* but can lead to timing failure instead of value. Any unexpected execution of an operation prior-to or within a time-sensitive event may cause the event to miss its deadline and generate timing-failure. Hence, such scenarios are classified as commission failure causing timing failure.

**Scenario 3.** When an unexpected change to any state value occurs.

**Scenario 4.** When an unexpected transition to other state occurs.

*Scenario 3* and *Scenario 4*, although unexpected, may or may not cause other failures. However, monitoring and reporting of such activities must be performed as it may lead to performance overhead within the system. Data reported from such events may help developers view the execution events and trace the process flow.

In some cases, unexpected execution of an operation does not interfere with other operations and no state or transitional changes occur within the component, then such execution could be ignored for failure. For example, in the Shopping Cart system, if the *Processing* component unexpectedly causes to compute the total amount for all items in the cart, it may not interfere with other operations such as adding or removing of shopping cart items. Such an execution causes no deviation in the software specification. Thus, no failure has occurred or reported.
The Analyzer module cannot provide definite results for some scenarios in commission failures, as it requires the knowledge of the system, which is difficult to translate into specifications. If the developer provides critical points where the commission failure may occur, he/she can eliminate the problems by fault prevention or fault removal techniques. Other techniques such as fault injection [11, 65], monitoring memory usage, or software profiling [66] may provide a better view of such failure anomalies. Since commission failure inherits scenarios from value and timing failures, detecting value or timing failure may provide hints on the cause of a commission failure. For example, if function X calls other function Y in a component, unexpected call to function Y from function X may cause it to produce timing failure. Hence, indirect analysis for detecting such failures could be performed. To detect such failures, the framework monitors all the events or function calls within a component. Execution records of all operations within a component will be analyzed when the component generates value failure, timing failure, or invalid state values or transitions. For all possible events that may have caused such failures, the Analyzer module will investigate timestamps of each execution records.

3.3.4 Omission Failure Scenarios

Omission failure identification is subject to similar issues as commission failure. Omission failure occurs when a component or an operation does not return a service or value when expected. Hence, it is necessary to monitor all the events within a component. Omission failures sometimes may be more damaging to the system than commission
failure as a function may expect results that other component may depend on. It is infrequent that failure of service (when required) from an operation can be ignored. The following presents the scenarios of omission failures:

**Scenario 1.** When no state change occurs as expected.

During the monitoring of state changes within a system, if no expected state change occurs, it can lead to omission failure. This is different from unexpected state change or when the component transitions to incorrect state. For example, consider a valid transition from state X to Y to Z. If a transition occurs from state X to Z, when expecting state X to Y leading to a particular failure, it is not classified as omission failure, but commission failure.

**Scenario 2.** When no result is returned by an operation and the operation deadline is missed.

The deadline missed due to lack of returned result is classified as timing failure caused by omission failure. For example, if a system is expecting results from a component but no result is returned due to the lack of connection or network issues, it could lead to an operation within a system’s component to miss its deadline and causing omission failure in accordance with Scenario 2.

**Scenario 3.** When a state change occurs within an operation but no result (success or failure) is returned by the operation.

**Scenario 4.** When an unexpected state change occurs and no result is returned by the operation. This could be considered as commission
and omission failure.

Scenario 5. When no result returned by an operation.

All these scenarios are variants of cases when omissions of results lead to other unexpected behavior. The return of failed result is not considered as omission but value failure. For example, in the Shopping Cart system, if the Processing component returns failure when processing cart for shipping, it is considered value failure. However, if the Processing component fails to return service, and no results (success or failure) are returned, it is considered as omission failure. From the above-mentioned omission scenarios (Scenarios 1-5), it is obvious that omission failure can also be considered as value failure (value with no result).

Identification of such scenarios is similar to commission failure detection. However, the detector cannot confirm the results of omission failures. In some cases, the results can be viewed by developer and could be classified as omission. While value and timing are identified, execution records of events within a component are analyzed for possible scenarios where omission failure caused incorrect values or missed-deadlines. These events are shown as possible causes of such failures and can be omitted from ranking if they are not definite.

3.4 Ranking Algorithm
Detecting failures and identifying the relevant components can help in many ways. For example, if failure $F$ was caused by a fault in component $C$, diagnosing and repairing the defect at the localized point could usually suffice to eliminate the otherwise threat of
failure $F$. The failure data is recorded into a repository, which will be used by the ranking algorithm to rank components. The ranking also provides an improved and systematic way to prioritize developer’s resources to mitigate failures. Such prioritization would allow choosing the first top-k components from the ranking output, which probably need more particular attention than other failures. We believe that the impact of choosing the “top-k” (failing) components can provide a good trade-off when compared to approaches, where all or random failing components are chosen.

For ranking components in given software under instrumentation, our algorithm requires ranking filters – type of failures (e.g., value failures) that are allowed to be included in the ranking process and whose values are set by a system developer. The algorithm, along with a number of failures, can use optional parameters such as QoS values to enhance the output set. The ranking algorithm has three main phases: i) Calculating Weighted QoS; ii) Comparing actual and acceptable failure rate; and iii) Overall ranking based on any or all of the four failure types.

3.4.1 Weighted QoS

Apart from using failure data to rank components, a software developer has an option to use QoS (Quality of Service) values to optimize component ranking. The developer, depending on the characteristics of a component, can set QoS values (e.g., cost, response time, or failure probability). The developer, based on the component’s functionality and the system’s service domain, must select its QoS attributes. For example, highly responsive components must have attributes such as mean time to failure, response time,
etc. For each QoS value, convex combination \([67]\) weights will be used, where the sum of all weights must be equal to 1.0. Weights are put on each parameter because we believe that not all parameters are of equal contributions for evaluating a component. Hence, QoS properties with higher weights will be given higher priority for calculating \(QoSRank\).

\(QoSRank\) is defined as follows:

\[
QoSRank(X) = \sum_{i=1}^{n} (q_i \times w_i) \tag{1}
\]

where \(q_i\) is the value for QoS type (e.g., cost, response-time.), and \(w_i\) is the weight for \(q_i\).

The \(QoSRank\) for each participating candidate component and corresponding comparable components are calculated. If the \(QoSRank\) of candidate component is lower than that of the lowest-valued comparable components, it is placed in the \(Priority\) set. Thus, QoS properties with higher weights will be given higher priority for calculating \(QoSRank\).

```
/* Distribution based on weighted QoS Rank */

Input: Participating candidate components, and their available comparable components
Output: Distributed in Priority or Secondary Ranking Set

For each candidate component participating {
    // Calculate QoS rank of candidate component and available comparable components
    QoSRank(Xc) = QoS Rank of candidate Component;
    minQoS = minimum(QoS Rank of all comparable components);

    if ( QoSRank(Xc) < minQoS ) {    // Candidate component is better than others
        putInPriorityRankingSet(Xc);    // Since the best in QoS Rank is still worse
    } else {                         // A comparable component is better
        putInSecondaryRankingSet(Xc); // Since the candidate could be replaced by the
                                         // non-monitored comparable one.
    }
}
```

**Figure 3.8: Calculating QoS Rank and arranging components in appropriate sets.**

Figure 3.8 uses the list of candidate components, which a developer chooses to be
included in weighted QoS ranking. The algorithm segregates the candidate components into Priority or Secondary set.

### 3.4.2 Acceptable Failure Rate

Acceptable failure rate feature of the ranking algorithm determines the acceptable failure rate for a candidate component using Equation 2. The equation allows a developer to input failure rates for specific failure types and corresponding weights to evaluate if the candidate component is within an acceptable failure rate.

$$\text{FailureRank}(X) = \sum_{i=1}^{n} (f_i \times w_i)$$ (2)

where $f_i$ is the failure rate of type $i$ (value, timing, commission, and omission) and $w_i$ is the corresponding weight. A developer can choose failure types based on the failures that are predicted for the particular component. Past component usage of failure history can also be used to evaluate failure type selection. For example, time sensitive components should mostly focus on calculating FailureRank based on timing failures. This can be achieved by having more weight on timing failure type.

```c
/* Distribution based on acceptable failure rate */

Input: Participating candidate components and their acceptable failure rates.
Output: Distributed in Priority or Secondary Ranking Set

For each candidate component participating {
    fR(Xc) = Calculate FailureRank of candidate component; // Actual failureRank
    aR(Xc) = acceptable failure rank set by the developer;
    if ( fR(Xc) < aR(Xc) ) {
        putInSecondaryRankingSet(Xc); // Failure rate is acceptable to the developer
    } else {
        putInPriorityRankingSet(Xc); // Xc is not acceptable
    }
}
```

**Figure 3.9: Calculating FailureRank.**
Figure 3.9 shows the implementation of acceptable failure rate algorithm. According to Figure 3.9, if the component’s actual \textit{FailureRank} is higher than the acceptable failure rate, the component fails on acceptable failure rate and it is placed in the priority set for developer’s higher attention. A developer can improve its reliability by applying appropriate mitigation reliability techniques. Finally, the overall ranking is computed by considering all candidate components. The weighted QoS and acceptable failure rate values are used to distribute components in either the priority or secondary sets.

\textbf{3.4.3 Overall Component Ranking}

The final phase of the ranking algorithm is to consider all participating components within a software system and use weighted QoS (Section 3.4.1) and acceptable failure rate (Section 3.4.2) to distribute components in priority or secondary sets. The components that are not chosen to participate in the first two sets will be in the third set where the components will be ranked based on the failures occurring within a component. Figure 3.10 shows how the components within each set are ranked.
Figure 3.10: Ranking each candidate component set based on the number of failures recorded.

In Figure 3.10, ranking inputs (from developers) are allowed to be included in the ranking process. For example, the developer can ask to include only value failures in Component X of the system. Hence, when calculating the total failures in Component X, only value failures will be calculated. The algorithm, based on filters and populated ranking sets, sorts each ranking set based on the collected failure data. The filters set by a developer are used to count numbers of failures. The components are prioritized within Priority and Secondary set as depicted in Figure 3.10. Figure 3.11 shows the overall process about how Priority set, Secondary set and the rest of the candidate components are ranked in the order of the required consideration for improving reliability.
In Figure 3.11, the first two phases of the ranking process are applied on required components, which will produce three sets – *Priority*, *Secondary*, and *Other*. Finally, these sets are ranked and sorted using the failure data and the functions described in Figure 3.10. Hence, based on component specifications, services provided, and failure data and ranking generated by the framework, a developer can assess the resources available, and use the technique that is best suited for a reliability goal.

### 3.5 Summary

The chapter described design assumptions, architecture and operation of the proposed framework. In addition, we described scenarios for each failure type that is detected using the framework. Finally, the chapter describes the implementation of the ranking...
algorithm and shows how the monitored data along with optional parameters such as weighted QoS and acceptable failure rate are used to rank candidate components.

The next chapter focuses on evaluating the framework’s effectiveness and performance overhead using two target systems.
Chapter 4

Experimental Evaluation

In this chapter, we evaluate our proposed approach by applying on two target systems. More specifically, Section 4.2 introduces the target systems under which the framework will be evaluated. Section 4.3 discusses the experimental setup for the evaluation of the proposed approach. Based on the experiments discussed in Section 4.3.1 and Section 4.3.2, we present our findings for the two target systems in Section 4.4. Section 4.5 summarizes the chapter.

4.1 Implementation Details

The current prototype is built in C/C++ and can be used to monitor software developed in the same programming language. Therefore, the framework requires standard C/C++ libraries, and GNU compiler (gcc or g++) [62] to compile and execute the framework integrated within the software being monitored. MySQL database [63] instance and C/C++ connector libraries are required for persistent data (failure records) repository.

4.2 Target Systems

We use two target systems to evaluate the effectiveness and performance of the proposed framework. An in-house developed system (i.e., a Shopping Cart system) and a popular open source versioning system, named CVS (Concurrent Versioning System) that will be used to show the scalability and use of the framework in a third party large scale target system.
4.2.1 Shopping Cart System

The primary reason to build this system is to quickly evaluate the possible features of the proposed framework before applying to other systems available in the open source community.

It is a custom C++ system with four major components. Events and operations within the components are designed so that all the four failure types can be evaluated and the components can be ranked accordingly. The in-house implemented system also helped us understand the complete target system and support the framework’s compatibility with a smaller target system. System specifications and critical states of the operation are readily available. Evaluating the framework on this target provides a better preview for the framework to be tested on larger systems.

4.2.2 Concurrent Versioning System

The Concurrent Versions System (also known as Concurrent Versioning System) is a revision control system for software development [31]. The system stores current version and other historical versions of a project. The system records the history of source files by enforcing site-specific policies. The source files to create binary can be found in many open-source repositories. However, not all files included in the source project are part of the core system files. For evaluation purposes, we will focus on core system components, which include 55 class files in the core source directory. This large-scale system comprises of 120,000 LOC but the experiments focus on approximately 30 different core components implemented in C (with 55,000 LOC). The components range from main.c
files to the implementation of popular features such as code `commit, update, edit, check-in, import, hash, patch, checkout` and others. To ensure that all features of our proposed framework are exploited during experimentation, we will modify some of the system specifications.

The CVS system was chosen due to its popularity in the open source community for version controlling and is also used in software industry. The system reliably stores implemented project data (and histories). It is important that the system ensures target requirements and handles restoring the project data as specified. Components that do not provide any value to the framework’s experimental evaluations were excluded from monitoring. The target system is a suitable match for our experiments as it is a component-based system that contains variety of events capable of generating different failure scenarios.

Table 4.1: Summary of the target systems for our experimental evaluation.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Description</th>
<th>Size (Lines of Code)</th>
<th>Number of core components</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shopping Cart</td>
<td>Shopping cart system for e-commerce application</td>
<td>1,500 LOC</td>
<td>4</td>
<td>C++</td>
</tr>
<tr>
<td>Concurrent Versioning System (CVS)</td>
<td>Versioning system for software development projects</td>
<td>55,000 LOC</td>
<td>27</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 4.1 provides the details of the size and the number of core components as well as the programming languages used for our subject systems. The core components are the ones that will be integrated with the proposed framework. Components that are not related to the core functions of the system are ignored in our experiments. For example,
the CVS system contains “diff” component that has no relation to any of the core features of the target system.

4.3 Experiments

In this section, we describe the experimental setup that is used to evaluate the effectiveness and performance of the framework. Also, the experimental setup and the execution process are discussed. Section 4.3.1 and Section 4.3.2 provide the overview, while the results based on the experiments we perform using the subject systems are mentioned in Section 4.4.

The experiments were performed on Macintosh OS X (10.7.4) with 2.26 GHz Intel Core 2 Duo processor and 4GB of DDR3 RAM. The development environment used for the code instrumentation was Apple XCode 4.3. To be consistent while compiling the source code for target executable, GNU GCC compiler was used. We also used the MySQL server [22] (version 5.3) with C++ connectors to store and retrieve persistent data that are used to rank components.

4.3.1 Framework Effectiveness

Here, we describe the experimental setup used to evaluate the effectiveness of the proposed framework in identifying failures and ranking components. Our ranking algorithm has optional parameters that further enhance the ranking of components by including weighted QoS values and acceptable failure rates. Since these values differ for every software developer and system specifications, our experiments will only emphasis on actual failure detection and ranking based on detected failures.
In order to create expected ranking of components for a target system, fault might be injected within a component by manual code instrumentation or by crafting different set of specifications where the implemented operation within a component would fail. Based on a constant operational profile and the number of executions for each participating operation within a component, the expected rank for each component is compared to the actual rank as evaluated by the framework. We assume that the fault produced within a component would lead to a failure of particular type. The results on success rate for failure detection as well as correct ranking order are provided in Section 4.4.1.

4.3.2 Performance Overhead
As with all monitoring applications, the use of APIs at the critical points of a target system creates extra overhead on the target system. The overhead caused by integrating our framework is directly proportional to the number of critical points identified and the monitoring APIs used. Integrating the proposed framework within the target system can create extra overhead on the computing resources and it not only increases the compiled processing instructions but also slows down the system. This overhead can be overlooked by the added benefits that the framework provides. The reason for added overhead is that the target source code has calls to the monitoring interfaces, which could possibly create added overhead on the system. Hence, it is essential to measure performance overhead of the underlying intended use of the system due to the integration of the framework.

For each participating target system, execution times of each participating component will be calculated. At each instance of the calculated execution time, the number of APIs
used during the experiment is also recorded. Hence, we can measure how change in the
number of critical points identified (and APIs used at those points) affects the
performance of the target system. After the results collection, execution times before and
after source instrumentation are measured to show the performance overhead caused due
to integration of the proposed framework on the target system. The native C functions are
used to calculate the execution times for each component. The function (that measures
the execution time) reports the resource usage (time spent executing user instruction) for
the specified process and the proposed framework stores the component’s execution
times.

The second experiment is meant to measure the change in the total execution time of
the target system if there is a change in the number of monitoring function calls. In this
experiment, for each target system, difference in execution times by increase in number
of APIs is measured. This provides an insight on the constant as well as variable
overhead caused by framework and its APIs. Percent change in execution time while
increasing number of APIs used within the target system is measured and tabulated in
Section 4.4.2.

4.4 Evaluation Results

In this section, we provide the results that were extracted by performing the
experiments described in Section 4.3.1 and Section 4.3.2. The experimental evaluation
for both Shopping Cart and CVS systems are performed and the collected results and
their synthesis are presented in this section.
4.4.1 Detection and Ranking

The effectiveness of the proposed failure detection and component ranking algorithm is based on the failure data gathered by the framework and developer’s input for QoS and failure rate parameters as mentioned in Section 3.4. During the experimentation, faults within the components were generated by (i) changing the specification provisions while keeping the implemented code and (ii) mutating the source code to make component deviate from the specifications. Using the operational profiles that are used during the experiments, the expected component ranking is noted and compared to the actual ranking as produced by the framework. For each failure type, a single run includes operating the target system under an operational profile where the actual versus the expected results as well as the probability of success in detecting failures are recorded. In total, 10 different runs were executed for each failure type to evaluate the ranking for each failure type. Table 4.2 shows the results from one such run.

<table>
<thead>
<tr>
<th>Ranking for failure type</th>
<th>Number of failures expected</th>
<th>Number of failures detected</th>
<th>Expected ranking (ascending order)</th>
<th>Actual ranking (ascending order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>29</td>
<td>25</td>
<td>[C1,C2,C4,C3]</td>
<td>[C1,C2,C4,C3]</td>
</tr>
<tr>
<td>Commission</td>
<td>41</td>
<td>46</td>
<td>[C1,C2,C3,C4]</td>
<td>[C1,C2,C3,C4]</td>
</tr>
<tr>
<td>Omission</td>
<td>12</td>
<td>9</td>
<td>[C2,C1,C3,C4]</td>
<td>[C2,C1,C4,C3]</td>
</tr>
<tr>
<td>Timing</td>
<td>43</td>
<td>37</td>
<td>[C4,C2,C1,C3]</td>
<td>[C4,C2,C1,C3]</td>
</tr>
<tr>
<td>All</td>
<td>124</td>
<td>116</td>
<td>[C2,C1,C3,C4]</td>
<td>[C2,C1,C3,C4]</td>
</tr>
</tbody>
</table>

In Table 4.2, for each failure type, faults are crafted within components of a target system. After the results produced by the framework, we verify the actual failures
detected by the framework. The expected ranking and the actual ranking are compared to evaluate the effectiveness of the framework detection and the ranking process. Each row compares specific failures detected and ranking order with the expected results. For example, row 5 shows that 116 out of 124 possible failures were detected resulting the actual ranking order as \([C2, C1, C3, C4]\). Here, the actual ranking is same as expected.

The results indicate that partial failure detection does not affect ranking by much as ranking process is relative (to other candidate components) while failure detection is independent of candidate components. However, in some cases (e.g., omission failure – row 3), it could affect ranking if inaccuracy in failure detection is so high that it affects relative ranking order of candidate components. Figure 4.1 shows the output produced by the framework for data shown in row 4 of Table 4.2.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Component Type</th>
<th>Failures</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority Ranked Components:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Processing</td>
<td>11</td>
<td>Timing: 1 Value: 1 Commission: 9 Omission: 0</td>
</tr>
<tr>
<td>Secondary Ranked Components:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Main</td>
<td>18</td>
<td>Timing: 1 Value: 0 Commission: 9 Omission: 0</td>
</tr>
<tr>
<td>Rest of Components:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Inventory</td>
<td>84</td>
<td>Timing: 7 Value: 0 Commission: 77 Omission: 0</td>
</tr>
<tr>
<td>4</td>
<td>Cart</td>
<td>3</td>
<td>Timing: 0 Value: 3 Commission: 0 Omission: 0</td>
</tr>
</tbody>
</table>

**Figure 4.1: Sample output result showing ranked components for the Shopping Cart system.**

As shown in Figure 4.1, the components participating in QoS and acceptable failure rate algorithm were classified in either *Priority* or *Secondary* ranked components (based on calculated data). This indicates that although *Inventory* component has the highest number of failures, *Processing* component should be considered first as it fails on QoS or
the acceptable failure rate property (or both). Inventory and Cart components were not part of the ranking based on QoS values and acceptable failure rate.

For each target system used in our experiments, the results were cataloged into 5 categories —four individual failure categories (value, timing, commission, and omission) and one consisting all failure types together. Ten runs on each category were executed and data was collected to evaluate the effectiveness of the proposed framework. Table 4.3 shows the effectiveness of the framework for each failure type as well as the overall effectiveness for the two target systems.

**Table 4.3: Percentage of failures detected and correct rank order for the target systems.**

<table>
<thead>
<tr>
<th>Target system</th>
<th>Failure type enabled to be detected</th>
<th>Percentage of failures detected by the framework (average of 10 runs)</th>
<th>Percentage of correct ranking produced by the framework (average of 10 runs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shopping Cart</td>
<td>Value</td>
<td>92%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>Commission</td>
<td>76%</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>Omission</td>
<td>72%</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>Timing</td>
<td>93%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>85%</td>
<td>90%</td>
</tr>
<tr>
<td>CVS</td>
<td>Value</td>
<td>86%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>Commission</td>
<td>75%</td>
<td>60%</td>
</tr>
<tr>
<td></td>
<td>Omission</td>
<td>72%</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>Timing</td>
<td>88%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>87%</td>
<td>80%</td>
</tr>
</tbody>
</table>

In Table 4.3, each row of the mentioned target system represents the average percentage of success when the actual result is compared with the expected result for each failure type. Each sample run for a particular failure type produces a ranking order that is either correct or incorrect. In some cases, our approach partially ranks components
(not 100% of the expected ranking). Such partially ranked order could still be useful to developers to apply mitigation techniques since mitigation on partially ranked components is better than the approaches where all or random components are selected. For some runs, a lower failure detection rate may still produce a correct ranking order as the failure in detection could be evenly distributed in all participating components. In addition, error in ranking due to QoS and failure rate set by a developer is not measured as it is considered as human error caused due to incorrect input. The framework is unable to detect all commission and omission failures, which indicates the need for more scenarios as well as limiting the scope of the detection. A higher number of success rates are observed when all failure types are considered and the result highly depends on the failure type detected at the critical point within the component. As the software complexity increases, it becomes difficult to extract specifications (especially from open source systems) and monitored components. The lack of domain knowledge and proper documentation can also restrict the success rate of failure detection.

4.4.2 Performance Overhead

In Tables Table 4.4 and Table 4.5, the execution times for each component within the target system are calculated for pre and post instrumentation of monitoring code. Tables Table 4.4 and Table 4.5 show the data gathered when the experiments are performed on the Shopping Cart system and the CVS target system, respectively. The execution time also depends upon the number and type of the monitoring APIs that are used. If the target system is small in size and requires very few resources to perform
operations and deliver services, integration of the framework produces more performance overhead. The integration of the proposed framework has two types of overhead: (i) A constant overhead that does not change with the number of APIs; (ii) The overhead caused due to APIs operations of the framework. The constant overhead may include initialization of Monitor module, creating commission or omission scenario objects generated by the framework to verify state values and transitions, file I/O calls, calls to monitor controller and the initialization of framework’s local repository. The overhead caused by such issues impacts the overall system performance but only occurs once during the integration of the framework.

Table 4.6 shows how the increase in monitoring calls escalates the execution time for each target system. This provides a better overview on the increase in performance overhead due to the monitoring calls. The overhead values are very noticeable when the framework is integrated with a smaller system that uses few computing resources. Table 4.5 shows the increase in execution times when the framework is integrated with a larger system, i.e., with the Concurrent Versioning System (CVS). The total core components involved in our experiments are mentioned in Table 4.1.
Table 4.4: Change in component execution times for the Shopping Cart system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Type of APIs used</th>
<th>Numb of APIs</th>
<th>LOC before instrumentation</th>
<th>LOC after instrumentation</th>
<th>Execution time before instrumentation (in seconds.)</th>
<th>Execution time after instrumentation (in seconds.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing</td>
<td>Timing</td>
<td>5</td>
<td>332</td>
<td>419</td>
<td>0.0002</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inventory</td>
<td>Timing</td>
<td>2</td>
<td>346</td>
<td>416</td>
<td>0.0002</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delegate</td>
<td>Timing</td>
<td>4</td>
<td>394</td>
<td>504</td>
<td>0.001</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cart</td>
<td>Timing</td>
<td>0</td>
<td>326</td>
<td>346</td>
<td>0.000053</td>
<td>0.0001</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.5: Change in component execution times for the CVS system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Type of APIs used</th>
<th>Number of APIs</th>
<th>LOC before instrumentation</th>
<th>LOC after instrumentation</th>
<th>Execution time before instrumentation (in seconds.)</th>
<th>Execution time after instrumentation (in seconds.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Timing</td>
<td>5</td>
<td>664</td>
<td>722</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkin</td>
<td>Timing</td>
<td>2</td>
<td>107</td>
<td>131</td>
<td>0.032</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkout</td>
<td>Timing</td>
<td>2</td>
<td>826</td>
<td>871</td>
<td>0.028</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>Timing</td>
<td>2</td>
<td>1648</td>
<td>1758</td>
<td>0.025</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>import</td>
<td>Timing</td>
<td>7</td>
<td>1279</td>
<td>1355</td>
<td>0.124</td>
<td>0.126</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>Timing</td>
<td>5</td>
<td>205</td>
<td>248</td>
<td>0.015</td>
<td>0.016</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>Timing</td>
<td>4</td>
<td>287</td>
<td>314</td>
<td>0.014</td>
<td>0.015</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>Timing</td>
<td>3</td>
<td>2023</td>
<td>2104</td>
<td>0.019</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commission &amp; Omission</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Notice that Table 4.5 (partially) shows the core candidate components in the experiment. Table 4.4 and Table 4.5 show participating components for target system along with the type of APIs used to detect specific failures within a component. The number of APIs used also accounts for the critical points identified within a component to detect the given failure type. The lines of code before and after instrumentation shows the change in the source code, which is the result of inserting the monitoring function calls at the identified critical points. The execution time before instrumentation provides a control time taken for the target system to execute without integrating the proposed framework, whereas the execution time after instrumentation shows the overhead (in terms of the execution time) caused due to the use of the framework. For example, Delegate component of the Shopping Cart system gained 110 lines of code after the instrumentation where 24 monitoring APIs were used (10 to detect omission and commission, 10 to detect value, and 4 to detect timing failures).

For each component, the execution calls before and after the framework integration shows decrease in performance caused by the framework. For each such component, the percent decrease in performance is not as significant as seen for the Shopping Cart system (Table 4.4) because the performance depends on the APIs used and is not proportional to the number of components (or system).
Table 4.6: Percent increase in execution time versus increase in number of APIs used for the target systems.

<table>
<thead>
<tr>
<th>Target System</th>
<th>Number of APIs – trial 1</th>
<th>Execution time – trial 1 (in seconds)</th>
<th>Number of APIs – trial 2</th>
<th>Execution time – trial 2 (in seconds)</th>
<th>% increase in execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shopping Cart</td>
<td>30</td>
<td>0.005</td>
<td>60</td>
<td>0.007</td>
<td>60.00 %</td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>30</td>
<td>0.005</td>
<td>91</td>
<td>0.012</td>
<td>168.99 %</td>
</tr>
<tr>
<td>CVS</td>
<td>124</td>
<td>0.312</td>
<td>200</td>
<td>0.335</td>
<td>23.1 %</td>
</tr>
<tr>
<td>CVS</td>
<td>124</td>
<td>0.312</td>
<td>289</td>
<td>0.462</td>
<td>48.08%</td>
</tr>
</tbody>
</table>

In Table 4.6, each row shows how the change in the number of APIs used to monitor the target system affects the total execution time of the system. For example, in the first row, doubling the number of APIs for the Shopping Cart system (from 30 to 60) does not double the overhead on the system. In the first row, 100% increase in API calls (from 30 to 60) causes only 60% increase in the execution time. Similarly, in the second row, 200% increase in API calls causes approximately 169% increase in the execution time indicating that some of the overhead caused by the framework is not exponential but linear added with some constant overhead as described earlier. The third and fourth rows show similar effect on the CVS system where it shows the percent of increase in the execution time for the CVS when the number of monitoring calls are increased. Table 4.4, Table 4.5, and Table 4.6 show that the performance overhead is a function of the number of APIs used. A decrease in performance of the target system is subjected to the interface functions called by the target system.
4.5 Summary

In this chapter, we discussed the experimental evaluation of the proposed framework and presented the evaluation results. We began by introducing the implementation details, and two target systems that were used for the experiments. Two experiments were performed to evaluate the framework: first experiment was to evaluate framework’s effectiveness and the second one was to evaluate its overhead.

Overall, our experimental results show the effectiveness in ranking components that can be used to systematically improve software reliability by using any mitigation techniques (at developer’s discretion). The results obtained by applying the framework into a target system can greatly benefit the developer during a maintenance process.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

Failure in a component within a system can lead to failure in delivering services that would affect a large number of users. Hence, critical components within a system must be identified and an appropriate reliability technique must be applied to mitigate component faults. In this thesis, we proposed a framework that monitors components within a system, detects failures based on specifications and ranks components by using failure data from the components and input from developers. Ranking can allow prioritizing components —and, therefore, maintenance— and can assist developers in choosing a systematic approach to improve software reliability.

We specified and implemented failure scenarios to detect specific failure types within a component. Once, the target system is instrumented for failure detection, runtime data is compared with specifications to analyze and classify failures. During post-runtime analysis, failure data and developer’s input such as QoS properties and acceptable failure rate are taken into consideration to rank components. This ranking process, by considering QoS properties from available components and acceptable failure rates of specific components can provide information on available resources and reveal service limitations to the developer. Developer’s input affects the ranking of components and may lower the maintenance priority if a better component is available or if a component has an adequate failure rate.
We conducted an experimental evaluation on two subject systems—i.e., the Shopping Cart system and the Concurrent Versioning system. The Shopping Cart system was developed in-house to cover and evaluate the features of the proposed framework. The Concurrent Versioning system was used to demonstrate the framework’s scalability. By evaluating the effectiveness and performance overhead, our results indicated that the framework could be a viable approach to prioritize components for effective maintenance of the target systems.

5.2 Limitations and Future Work

Our approach has three major limitations. First, it requires manual identification of critical points within a component and inserting monitoring code to detect failure. By identifying operations (by the framework) within a component and let developers select critical points where the framework can insert monitoring code could be a way-out to deal with such a limitation. Performing static analysis on the source code can identify operations within a component. Once the operations are identified, responsibility for critical point selection lies in the hands of the developer. This would also provide a semi-automated process of instrumenting the monitoring code without creating any additional overhead by inserting APIs that may not be required.

The second limitation is the fact that our framework performs post-runtime analysis to classify failures and rank components. We plan to explore more effective way to analyze failures and perform real-time ranking based on data collected during execution. This could provide a time-based graph on how and when the components fails during its
execution and offer better insight on the underlying causes and appropriate mitigation techniques. It also expands the coverage of the framework beyond off-site testing and component ranking to improve software reliability.

Finally, the current failure scenarios are limited to four failure types on certain scenarios. This could be improved by considering more failure scenarios for each failure type. This could expand failure detection, ultimately can improve the ranking process. Also, other factors such as invocation frequencies and the target system architecture could also be taken into account when ranking components.
References


