Abstract

Depending on the role of software in everyday life, the cost of a software failure can sometimes be unaffordable. During system execution, errors may occur in system components and failures may be manifested due to these errors. These errors differ with respect to their effects on system behavior and consequent failure manifestation manners. Predicting failures before their manifestation is important to assure system resilience. It helps avoid the cost of failures and enables systems to perform corrective actions prior to failure occurrences. However, effective runtime error detection and failure prediction techniques encounter a prohibitive challenge with respect to the control flow representation of large software systems with intricate control flow structures. In this thesis, we provide a technique for failure prediction from runtime errors of large software systems. Aiming to avoid the possible difficulties and inaccuracies of the existing Control Flow Graph (CFG) structures, we first propose a Connection Dependence Graph (CDG) for control flow representation of large software systems. We describe the CDG structure and explain how to derive it from program source code. Second, we utilize the proposed CDG to provide a connection-based signature approach for control flow error detection. We describe the monitor structure and present the error checking algorithm. Finally, we utilize the detected errors and erroneous state parameters to predict failure occurrences and modes during system runtime. We craft runtime signatures based on these errors and state parameters. Using system error and failure history, we determine a predictive
function (an estimator) for each failure mode based on these signatures. Our experimental evaluation for these techniques uses a large open-source software (PostgreSQL 8.4.4 database system). The results show highly efficient control flow representation, error detection, and failure prediction techniques. This work contributes to software reliability by providing a simple and accurate control flow representation and utilizing it to detect runtime errors and predict failure occurrences and modes with high accuracy.
I would like to thank my supervisor Dr. Mohammad Zulkernine for all the support to complete my PhD thesis. I also would like to thank Dr. Pat Martin and Dr. Hossam Hassanein for their strenuous efforts and wonderful support during my years of study. I sincerely thank Dr. Selim Akl, Dr. Jim Cordy, and Dr. Robert Crawford for their positive contributions to build and strengthen my academic expertise and profession. I am also thankful and grateful to my thesis examination committee: Dr. Ivica Crnkovic and Dr. Thomas Dean for their encouragement and enhancing reviews to my theses. I am also grateful to all the members of Queens Reliable Software Technology (QRST) Lab with whom I had numerous thoughtful discussions.
Statement of Originality

I hereby certify that this thesis is the original work of the author. Ideas and techniques contributed by others are fully acknowledged in accordance with the standard referencing practices.

Atif S Mohamed

June, 2012
## Contents

Abstract i

Acknowledgments iii

Statement of Originality iv

Contents v

List of Tables viii

List of Figures x

Chapter 1: Introduction 1

1.1 Motivations 2

1.2 Contributions 4

1.3 Work Overview 8

1.4 Organization 8

Chapter 2: Background and Related Work 10

2.1 Basic Terminology and Definitions 10

2.1.1 Software Components and Connectors 10

2.1.2 Reliability Means and Impairments 12

2.1.3 Control Flow Graph (CFG) 13

2.2 Related Work 14

2.2.1 Control Flow Representation 14

2.2.2 Control Flow Monitor 19

2.2.3 Failure Mode Prediction 24

2.3 Summary 27

Chapter 3: A Control Flow Representation Technique 29

3.1 Overview 30

3.2 An Illustrative Example 30

3.3 Complexity and Inaccuracy of the Existing CFGs 33

3.3.1 Complexity 35

3.3.2 Inaccuracy 36
5.2.6 Performance Overhead .............................................. 99
5.2.7 Regression and Prediction Processes Summary ............... 100
5.3 Experimental Evaluation ............................................. 101
  5.3.1 Experimental Process and Setup ................................. 102
  5.3.2 Actual Failure Occurrences and Modes in System Runs .... 104
  5.3.3 Example Error Log Trails ....................................... 105
  5.3.4 Correlation and Prediction Error Measures ................... 107
  5.3.5 Predicted Failure Mode Eventualities Along System Runtime 109
  5.3.6 Failure Mode Prediction in System Runs ...................... 112
  5.3.7 Comparing Actual and Predicted Failure Results ............. 113
5.4 Summary ................................................................. 115

Chapter 6: Conclusions, Limitations, and Future Work 117
  6.1 Conclusions .......................................................... 117
      6.1.1 Control Flow Representation ................................ 117
      6.1.2 Control Flow Monitor ........................................ 119
      6.1.3 Failure Prediction ........................................... 120
  6.2 Limitations and Future Work ...................................... 122
      6.2.1 Control Flow Representation ................................ 122
      6.2.2 Control Flow Monitor ........................................ 122
      6.2.3 Failure Prediction ........................................... 123

Bibliography 124

Appendix A: Sample Inputs/Outputs of the Proposed Techniques 139
  A.1 XML Representation ................................................ 139
      A.1.1 XML Headers .................................................. 139
      A.1.2 XML Documents .............................................. 141
  A.2 Control Flow Monitor ............................................. 143
      A.2.1 Signature Injection .......................................... 143
      A.2.2 Automatic Execution Toolkit ................................. 145
      A.2.3 Exported Runtime Signatures ................................. 147
  A.3 Failure Mode Prediction .......................................... 148
List of Tables

3.1 The software architecture representation of the code example of Figure 3.2. 43
3.2 Architectural patterns, nesting relations, and examples. 53
3.3 The state transfers of Figure 3.8, their conditions, and existence in the architectural patterns. 57
3.4 Transition groups of the code example of Figure 3.2. 59
3.5 The information fields derived during the steps of rendering the CFGs. 60
4.1 The signatures and injection locations of the example code of Figure 3.2. 75
5.1 The possible error patterns determined using the error state parameters. 92
5.2 The error counters, error spread-signatures, and failure mode eventuality measures of the example runtime log shown in Figure A.10. 95
5.3 The system runs utilized in the regression and prediction steps, their experimental data sets, and utilized system versions. 102
5.4 Description of the failure modes that have occurred in the systems runs. 104
5.5 The actual failure mode occurrences in systems runs. 108
5.6 The correlation coefficients and the regression square errors averaged among the failure mode prediction functions. 107
5.7 The failure mode eventualities of system runs with respect to old and new error log trails. 111
5.8 Failure mode eventualities in the old and the new log trails of system runs. 113
5.9 Prediction accuracy of failure occurrence and mode. 115
A.1 The actual execution times of the systems runs against the experimental

data sets. .......................... .......................... 149
List of Figures

1.1 Overview of the thesis techniques. ........................................ 8

3.1 Overview of the CDG derivation method. ............................. 30
3.2 An example C code for Depth First Search (DFS) algorithm. .... 31
3.3 The CFG structures of the provided code example of Figure 3.2. .. 32
3.4 Types of Control Flow Graph (CFG) edge representations. ......... 34
3.5 A CDG graph representing the example code of Figure 3.2. ......... 40
3.6 Intermediate tree data structure for representing a software architecture. . 47
3.7 The XML representation of the example code of Figure 3.2. ......... 50
3.8 A state diagram describing the control flow among the specified transition groups. ....................................................... 56
3.9 A Depth First Search (DFS) algorithm for deriving a program CDG. . 61
3.10 The numbers of nodes and edges of the CFG structures representing program segments with variables sizes. ............................ 66
3.11 The occurrences of the edge types in the control flow graphs. ......... 66
3.12 The numbers of edges of each CFG structure and edge type averaged over the different program segments. ............................. 67
3.13 The numbers of illegal control flow branches in the CFGs representing the program segments of variable sizes. ............................ 68

4.1 Overview of the control flow error detection mechanism. ............ 71
4.2 The CDG representation of the program partitions representing the code example of Figure 3.2. .................................................... 73
4.3 The control flow monitor structure. ......................................... 76
4.4 The control flow error detector. ............................................ 79
4.5 The control flow errors in versions and runs. ......................... 83
4.6 The control flow error types, characteristics, and highest occurrences in system components. .................................................. 85

5.1 Overview of the failure occurrence and mode prediction approach. .... 89
5.2 An illustrative example of the predicted eventuality measures correspond- ing to an error log record. ..................................................... 94
5.3 Example distributions of the signature variables along the runtime intervals of different runs and experiments. ................................. 106
5.4 The correlation coefficients of the failure mode estimators. ................ 107
5.5 Example correct predictions of success and failure runs with old log trails. 109
5.6 Example correct and incorrect predictions of success and failure runs with new log trails. ......................................................... 110
5.7 Example incorrect predictions of success and failure runs with old and new log trails. ........................................................... 111

A.1 The header file of the XML document representing the program structure. 139
A.2 The header file of the XML document representing the program components. 140
A.3 PostgreSQL version 8.4.4 system structure. ................................. 141
A.4 The structure of a sample component (pg_dump) of the PostgreSQL project. 142
A.5 Automatically injected signatures in the PostgreSQL database software. . . 143
A.6 The procedure dOSpy() that exports the runtime signatures to the output stream. .............................................................. 144
A.7  Part of the shell script of the automatic operation toolkit. . . . . . . . . . . 145

A.8  A sample SQL file that is passed as a parameter to the automatic execution
toolkit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146

A.9  The exported runtime signatures by an executed version of the PostgreSQL
database. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147

A.10 The error results of a sample system run of the PostgreSQL database. . . 148
Chapter 1

Introduction

Contemporary software systems aim to efficiently provide more rigorous functionalities using more sophisticated architectures. The integration among software enterprises and their continuous sprawl across the Web infrastructures enabled the spread of large scale software systems in numerous aspects of our daily lives. Software quality research and practices are striving to keep pace with the technology transitions with respect to these system scales. Reliability (the continuity of correct service) is the first of five important software quality characteristics defined by the Consortium for IT Software Quality (CISQ) [1]. Depending on the software role in everyday life, the cost of unreliable software can sometimes be unaffordable. Strenuous efforts are needed for reconstructing reliability mechanisms to suit these systems from the traditional reliability theory. This thesis focuses on the reliability of contemporary software systems by presenting a number of techniques that facilitate failure prediction from runtime errors in large software systems. In the following sections, we discuss our motivations and contributions. Then we provide the thesis work overview and describe the thesis organization.
1.1 Motivations

Practical analysis of failure manifestation in a software system involves tracking the underlying causes (errors) and sources (faults). Different failure manifestation manners can be characterized by the error distribution (across system components or among the error parameters) and increase along a system runtime interval. The consideration of failure manifestation process in reliability analysis techniques is a vital goal for developing future software engineering paradigms [2]. Detecting runtime errors and predicting failures in an early stage of their manifestation is important to assure system resilience. It helps avoid the costs of failures and enables systems to perform corrective actions prior to their occurrences.

Failure prediction methods have been used by leading software manufacturers (IBM [3] and Microsoft [4]) to minimize the impacts of failures on product performance and the cost of operation risks based on the code changes in product releases. Large industrial and research organizations devote considerable efforts and budgets to analyze failure modes and their effects in software systems. A failure is a deviation from an intended function [5] and a failure mode is a manner of failure manifestation [6]. For example, the National Aeronautics and Space Administration (NASA) uses a number of reliability practices in several space programs including Apollo, Viking, Voyager, and Galileo to analyze the failure mode behavior of their systems [7, 8]. These practices have also been extensively used in aerospace, nuclear, automotive and medical industries to identify the possible failure hazards of the embedded systems in their products [9]. Existing failure prediction techniques involve forecasting failure counts of system runtime intervals [10, 11], failure effects [12–17], and failure occurrences in the near execution future [18–23]. Both failure count and failure effect prediction techniques do not forecast failures during system runtime. Instead, they forecast failure numbers and estimate the effects of possible failure
modes that may occur in future time intervals. However, existing failure occurrence prediction techniques do not estimate the failure modes and do not determine any probability measures of failure occurrences.

As an underlying technique for failure prediction, runtime error detection encounters a prohibitive challenge with respect to the control flow representation of large software systems with intricate structures\(^1\). Some existing techniques use finite state automata [24] or branch traces [25,26] to represent software control flow. However, the majority [27–68] use Control Flow Graphs (CFGs) for this purpose. Several CFG structures are proposed in the current literature and some of them are intensively used in software reliability analysis techniques. Three CFG structures are mainly distinguished in these techniques: Statement-based (S-CFG), Block-based (B-CFG), and Component-based (C-CFG) control flow graphs. These CFGs are either remarkably complex or lack accuracy by allowing some illegal control flow sequences in their representations. The S-CFG is congested with edges representing control transitions among all program statements regardless of their contents and locations in the code structure. The latter two CFGs (B-CFG and C-CFG) aim to avoid this congestion by selectively widening the node representation to code blocks and components, respectively. However, based on these node selections, the CFGs consider a tradeoff between the complexity and the accuracy in their representations. Due to the possible inaccuracies of control flow representations, reliability analysis techniques may encounter improper information about the internal system states and other consequent flaws in their results. Moreover, maintaining simple control flow representation is important to practically implement these analysis techniques in large software systems.

Error detection techniques mainly use signature-based approaches to monitor the control transitions among code instructions, system components, or basic blocks and detect

\(^1\)Structure and architecture are sometimes used interchangeably in this thesis.
1.2. CONTRIBUTIONS

their anomalies. These techniques detect control flow errors by the use of watchdog, redundancy, assertion, or monitoring approaches. These approaches vary among each other by trading-off among four main goals: avoiding performance overhead, avoiding program modifications, avoiding the use of extra hardware, and increasing error coverage by detecting more errors of different types or among different scopes of control transitions. Signature-based control flow monitors can save more performance overhead than the other techniques and at the same time they do not require any additional hardware or software redundancy. Based on the signature structure, these techniques can also increase the error coverage. Existing signature-based approaches rely on the B-CFG to represent the system control transitions [27, 28, 57–68]. Therefore, in these techniques, validating the control flow transitions may not be feasible for large software systems and may be susceptible to false negatives due to the inclusion of some illegal control flow branches in the B-CFG.

1.2 Contributions

The contributions of this thesis are integrated towards building effective failure prediction from system runtime errors. Three main techniques are provided to achieve this thesis goal: connection dependence graph for control flow representation, connection-based signature approach for control flow error detection, and failure occurrence and mode prediction from system runtime error logs.

The Connection Dependence Graph (CDG) contributes to the efficiency of architecture-based reliability analysis by avoiding the complexities and inaccuracies of the underlying CFG structures in the existing techniques. A CDG is a connection-based control flow graph representation of all possible connection sequences that might be traversed through a program during its execution. We describe the CDG and explain how to derive it from program source code. In this derivation, a number of architectural patterns (groups of
statement types with similar control branching structure) and a number of transitional patterns (groups of control transitions with similar structure) are defined and exploited to capture the control transitions and build the CDG.

The CDG facilitates the control flow representation of large software systems and helps avoid the possible flaws of illegal control flow sequences and their consequent inaccuracy in system reliability analysis results. Like other CFG structures, a CDG can be an essential tool for software reliability techniques in research and practice. A few examples of these techniques are static analysis, dynamic analysis, software instrumentation and injection, reliability assessment, and control flow error detection. In these techniques, a CDG can be used not only for program analysis purposes but also for constructing software reliability approaches. Relative to the existing CFG structures, the use of the CDG in reliability analysis techniques may considerably decrease the state space size and simplify the mathematical derivations. Moreover, the CDG is not limited to the inter-component control transitions and its representation outlines comprehensive details about the intra-component control flow transitions. Furthermore, the CDG preserves the connection invocation dependencies (deterministic invocation order) in the control flow representation. Consequently, the use of the CDG in reliability analysis helps achieve more rigorous reliability analysis of software system paradigms.

The CDG differs from the existing control dependence graphs by being a lightweight CFG representation with simpler dependence relationships among graph nodes. Moreover, it can be derived directly from software architecture rather than from other CFG structures. The proposed CDG can be used during the design phase to represent software architecture and provide graphical transformation of logic and data requirements. During the testing phase, the CDG can also be used in static analysis techniques to validate the software implementations against the specified software architecture. Dynamic analysis
can utilize the CDG at the operational phase to observe and determine the behavioral properties of software systems. It can also be used to represent system state transitions for quantifying software system reliability and other quality attributes.

We utilize a CDG representation to construct a connection-based signature approach for control flow error detection. We first describe our connection-based control flow signature structure in which, we partition the program components into code blocks. A connection-based control flow signature is a runtime state describing the currently invoked connection during a system runtime. Each partition (or block) is associated with a CDG to represent the control structure of its code content. Next, we provide our control flow monitor structure and error checking algorithm based on these CDGs. Our signature-based control flow monitor does not require any additional hardware or software redundancy. The underlying control flow representation using CDG in our monitor decreases the possibility of false negatives due to its higher accuracy relative to the commonly used B-CFG representation.

Our connection-based signature structure allows complete coverage of all program connections and enables the tracking of their invocations and return addresses. Therefore, it can relate a runtime state to the execution traces of a software system in order to determine control flow errors with respect to the unreturned connections (existing in the control stack) in case of improper program termination. Furthermore, the monitor helps locate the responsible component of each control flow error. It further narrows down the location to the code statement at which an illegal branch or unreturned connection has taken place. The use of the CDG in control flow error detection allows injecting only one line of code with respect to each graph node during a program instrumentation. Given that, the CDG can significantly reduce the performance overhead of a control flow monitor in comparison to the existing techniques that inject two lines in each block. Our control
flow error detection technique can be used to monitor the system behavior in computing environments where operational faults may cause control flow errors.

We utilize the detected errors and state parameters to predict the possible consequent failure occurrences and modes during system runtime. In a form of error log, we use these errors and erroneous state parameters to craft error spread-signatures that summarize error occurrences throughout system runtime interval. An error spread-signature is a runtime state describing the error distribution among erroneous state parameters and error increase during a system run. Using system error log history, we determine a predictive function (an estimator) for each failure mode based on these signatures. A failure mode eventuality measure indicates the probability of occurrence of a failure mode in a system runtime. We show how to use these estimators and resolve their results to determine the predicted failure occurrence and mode in correspondence to each log record. The main contributions of this work involve predicting the failure eventuality during system runtime and forecasting the possible failure modes. Predicting failures at runtime enables autonomic systems to perform corrective actions prior to failure occurrences. This technique enables the practical consideration of different criticality levels or severities of different failure modes in the reliability analysis. The proposed failure prediction approach is an operational phase technique. It exploits runtime error logs to predict failure occurrences and modes prior to their manifestation.

To summarize, the proposed techniques in this thesis contribute to software reliability by providing a simple and accurate control flow representation and utilizing it to detect runtime errors and predict failure occurrences and modes with high accuracy.
1.3 Work Overview

Figure 1.1 provides an overview of the proposed techniques and illustrates their relationships. Three techniques are shown as rectangular shapes and their inputs and outputs are shown as document shapes. A dashed rectangular box indicates the software system and contains some of its aspects that are utilized in this thesis: code, runtime history, and runtime operation. The first technique (control flow representation) uses the software system source code to generate a CDG corresponding to the program structure. The second technique (control flow monitor) uses the program CDG to validate the system runtime signatures and generate an error log. Finally, the third technique (failure mode prediction) uses the system runtime history and current runtime error log to forecast the possible failure occurrences and modes. The three techniques are described in details in Chapters 3, 4, and 5, respectively.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 provides relevant background information and definitions of software reliability and architectures. It also describes our related work with respect to the proposed techniques. In Chapter 3, we introduce

\[^2\text{The error spread-signature is different from the control flow signature described earlier.} \]
our control flow graph representation. We describe the CDG and explain its derivation methodology. Then, we provide an experimental evaluation of the approach. Chapter 4 describes our control flow monitor. We present the control flow monitor structure and explain the error checking algorithm. Then, we evaluate the approach. In Chapter 5, we present our failure prediction approach by describing the error spread-signature and the failure mode eventuality. We derive an eventuality estimator for each known failure mode and show how these estimators can be used to predict failures. We provide an experimental approach for evaluating the accuracy of failure prediction. Finally, in Chapter 6, we discuss the work limitations, future work, and conclusions.
Chapter 2

Background and Related Work

Prior to proceeding in this thesis, we state some basic terminology about software architecture and system reliability. Then, we discuss the background and related work of the proposed techniques.

2.1 Basic Terminology and Definitions

In this section, we describe the main terminology in software architecture research and control flow graph representation that are used throughout this thesis. Then, we define and explain the means and impairments of software system reliability.

2.1.1 Software Components and Connectors

*Software architecture* is the structure, which comprises software components, externally visible properties of those components, and relationships among them [69]. In the current literature, the definition of a software component is usually tailored based on the architectural style, programming paradigm, or system paradigm [70]. In Component-Based Software Development (CBSD), a component is defined as a unit of composition and deployment with contractually specified interfaces, explicit context dependencies only, and no persistent state [71, 72]. Examples of component forms in current software systems
include function [45], module [73], class [74], distributed agent [75], and model-based component [76]. Due to the diversity and inconsistency of the existing component definitions, we define a software system component as a source code and data container that has identified boundaries and specified interfaces with other containers that together construct a software system.

A connection is defined as a mean by which a component interacts with another component (e.g., procedure call, event trigger, and data access). A connection represents a control flow and/or data transfer channel between two architectural points: connection invoker and implementer. At the architectural level, a connection invoker represents an input port and a connection implementer represents an output port of a component interface. At the program code level, a connection invoker (or simply, a connector) and a connection implementer are specified based on the programming paradigm. The connectors involve procedure calls, events, and data accesses [77]. Unlike data access connectors, the procedure call and the event connectors allow synchronous and asynchronous control transfer, respectively. A procedure call connector models the flow of control between two components and allows data transfer between them through the use of formal parameters. An event is an instantaneous initiation or termination of a process that is handled by another or by the same software component.

A connection implementer expresses a procedure declaration statement, an event handler, or data object. For the former two connectors (procedure call and event trigger), a Connection Implementation Block (CIB) is a portion of program code that represents the body of the connection implementer and that is executed each time one of its connectors is visited. For example, in the procedural programming paradigm, a CIB is equivalent to a procedure body and a connector is a procedure call. In object oriented programming, a CIB is a class method including constructors and destructors and a connector is an implicit method call or an automatic invocation of a class constructor or destructor.
In event-driven programming paradigm, a CIB is an event handler and a connector is an event trigger. A *connection invocation* is the process of control and/or data transfer from the connection invoker to its CIB. A *connection return* is the process of control and/or data transfer from the CIB back to the connection invoker. Relative to a connector, we refer to the containing CIB as Invocation Code Block (ICB). Therefore, an ICB is a code block where a connection is invoked (the caller block) and a CIB is a code block where a connection is implemented (the callee block).

### 2.1.2 Reliability Means and Impairments

Faults, errors, and failures are the main impairments to software reliability. A *fault* is a defect or bug in the code which can cause errors. An *error* is a system state which may lead to failure. A *failure* is a deviation from an intended function. A *failure mode* is a manner of failure manifestation [6]. A failure mode also specifies a class of failure hazard. Failures are classified according to the impact domain into content, silent, early service delivery, performance, halt, and erratic failures. [78].

*Software system reliability* is the probability that a system performs its intended function for a specified period of time under a set of specified environmental conditions [5]. In general, the means to achieve software reliability are fault avoidance (or prevention), fault removal, fault/failure forecasting, and fault tolerance. Fault avoidance or prevention techniques are employed in software development to reduce the number of faults introduced during the software construction. Fault removal techniques are employed during software verification and validation for detecting existing faults and eliminating them. Fault (or failure) forecasting estimates the presence of faults and their failure consequences during the design, validation, and operational phases. Fault-tolerance provides services complying with the specification in spite of design and operational faults by utilizing some kinds of redundancy.
2.1.3 Control Flow Graph (CFG)

In its general form, a control flow graph (CFG) is a representation of all possible execution paths that might be traversed through a program during its operation. The graph nodes represent the basic code blocks and the graph edges represent the control flow transitions among these blocks. In most presentations, there are two designated virtual nodes: the start node and the end node where the control enters and leaves the control flow graph, respectively. A control flow sequence is a number of connected nodes. We define a control flow branch as a special sequence of three nodes and two edges connecting them. Control flow branches are especially important in this thesis as we use them to measure the inaccuracy of a CFG. A path between two nodes is a sequence of edges that connects these nodes in a depth first (DFS) traversal of the graph. Some of the graph nodes are characterized according to their relations with other nodes. In a specific path, a node 1 is an ancestor of a node 2 and 2 is a successor of 1, if 1 precedes 2 in the node sequences of this path.

In a CFG, we define an ignored connection as a procedure call or event trigger that is not represented by a CFG node due to the graph structure (e.g., S-CFG, B-CFG, and C-CFG). We also define a supplementary edge as a CFG arc that bridges an ignored connection by connecting the preceding node(s) to the associated CIB in order to express the invocation process and by connecting the CIB to the succeeding node(s) of the ignored connection to represent the control return. For example, consider three successive statements 1, 2, and 3 of program code where statement 2 includes a procedure call $r$. If (in some CFG structures), statements 1 and 3 are associated with CFG nodes ($n$ and $m$, respectively) while statement 2 is not, then some supplementary edges are introduced to represent the procedure call $r$. These supplementary edges are the arcs from the node $n$ to the CIB of $r$ and the arcs from this CIB back to $m$. An illegal control flow branch indicates two successive transitions where the first transition is correct (i.e., complies with the
software architecture) while the second one is incorrect. An illegal control flow sequence is any sequence of transitions that includes an illegal branch. A control flow transition is incorrect, if its occurrence during system runtime makes a current legal control flow sequence to be illegal.

2.2 Related Work

The work in this thesis includes control flow representation, control flow error detection, and runtime failure prediction techniques. In the following subsections, we compare and contrast our proposed techniques with the existing work.

2.2.1 Control Flow Representation

Control flow diagrams/graphs have been extensively used in static analysis to detect poor and potentially incorrect program structures and in dynamic analysis to examine the behavioral properties of programs. They represent the logic and data requirements, design, or code using graphical views which are easier to examine and utilize in system engineering techniques [79]. A control flow diagram (CFD) describes the control flow structures among the high level objects of a system (e.g., business processes, classes, components, or subprograms). A control flow graph (CFG) is a special diagram that represents all paths that might be traversed through a program during its execution.

The CFG has been intensively used in several areas of software engineering disciplines including architectural design, program analysis and compilation, and software quality analysis and measurement. Software reliability analysis investigates reliability impairments during program development and operational phases. Some existing techniques use finite state automata [24] or branch traces [25, 26] to represent software control flow. However, the majority [27–68] use CFGs for this purpose.
A CFG structure is mainly characterized by the graph node and edge representations. Three CFG structures are mainly distinguished in current reliability analysis work: S-CFG, B-CFG, and C-CFG. In these CFGs, edges represent the control transitions among graph nodes (statements, blocks, or components). Unfortunately, these three CFG structures are either remarkably complex or lack accuracy by allowing some illegal control flow sequences in their representations. In the following paragraphs, we describe some example usages of these CFG structures and highlight their limitations, then we describe some research attempts to construct other CFG structures.

The S-CFG represents each program code statement as a graph node [33–36]. In these techniques, a CFG is used to calculate inter-procedural control dependencies [33], calculate execution paths’ reliability [34], pinpoint sources of errors [35], and construct system reliability model [36]. As underlying control flow representation of these techniques, the S-CFG can be congested with edges representing control transitions among statements regardless of their contents or locations in the code structure. In comparison to the S-CFG, the proposed Connection Dependence Graph (CDG) is less complex since it relatively includes fewer numbers of nodes and edges. Thus, relative to the S-CFG, the use of the CDG in these techniques may considerably decrease the state space sizes and simplify the mathematical derivations.

The B-CFG partitions a software program into branch-free code blocks and represents them as graph nodes [27, 28, 57–61, 63–68]. These branch-free code blocks are usually referred to as basic blocks. In these techniques, a branch-free block is defined as a maximal set of ordered code instructions in which the execution starts at the first and ends at the last statement. A block cannot include any control flow branch with an exception only for the last instruction [63]. A control flow branch can be caused by a jump instruction, a procedure call, or a control structure instruction [66]. The basic blocks can be determined
2.2. RELATED WORK

at the level of the assembly code or at the high-level programming language code [80]. Alkhalifa et al. [27] utilize the B-CFG to evaluate the capabilities of system-level control flow error detection mechanisms. Borin et al. [28] classify control flow errors based on the B-CFG and implement a comprehensive error detection technique based on this classification. Wu et al. [67] aim to increase the error coverage of control flow checking techniques. The use of a CDG rather than a B-CFG in error detection techniques allows complete coverage of all program connections. Unlike B-CFG, the use of the CDG in these techniques can identify the error location at the statement level rather than the code block level. Furthermore, unlike B-CFG, using CDG in control flow error detection allows injecting only one line of code with respect to each graph node. Given that, the CDG can significantly reduce the performance overhead of a control flow monitor in comparison to the existing techniques that inject at least two lines in each block.

The C-CFG represents each program component as a graph node [29–32,37–40,42–47, 49–56]. Cheung et al. [37] use the C-CFG to represent software architectural models and associate stochastic approaches for system reliability modeling early in the development lifecycle. Hamlet et al. [38] produce software component measurements and use them to calculate composite system reliability with the aid of C-CFG. Gokhale et al. [52] provide a unifying framework for state-based models for architecture-based software reliability prediction. They use the C-CFG to represent a discrete time Markov chain (DTMC) and other stochastic models for inferring the reliability measures. In general, the C-CFG structure avoids the complexities of S-CFG and B-CFG, by selectively widening the node representation to components instead of statements and blocks. However, this node selection may decrease the representation accuracy by disregarding the intra-component control flow transitions as well as the inter-component connection invocation dependencies (deterministic execution orders). Any graph node that has dependent outward edges to other nodes is considered a possible source of illegal control flow sequences in this
2.2. RELATED WORK

Due to an inaccurate control flow representation, the aforementioned reliability analysis techniques may encounter improper information about the internal system states and other consequent flaws or blemished results. The CDG is not limited to the inter-component control transitions and its representation outlines comprehensive details about the intra-component control flow structures. Moreover, the CDG preserves the connection invocation dependencies among software components in the control flow representation. Due to its higher accuracy relative to the C-CFG, the CDG avoids the possible flaws of illegal control flow sequences and their effects on the accuracy of reliability analysis techniques. Consequently, the use of the CDG helps achieve rigorous reliability analysis of software system paradigms. Similar to the C-CFG, the simplicity of the CDG can also decrease the state space size and simplify the mathematical derivations.

Some research work attempted to construct different CFG structures to decrease the representation complexity or to consider the control dependence among program statements. Stafford et al. [45] classify the CFG structures that provide special considerations to procedure calls in their architectures into in-lined and one-to-one control flow graph representations. An in-lined CFG is constructed by inserting individual copies of each procedure’s control flow graph after each call to this procedure [33]. A one-to-one CFG uses only one copy of a procedure’s CFG and inserts control flow arcs between each procedure call and the appropriate procedure’s CFG to represent the call to, and the return from, the procedure [81–83]. However, these CFG representations mainly aim to extend the control dependence identification models from uni-procedure to multi-procedure programs by considering procedure calls. However, our connection dependence graph does not only focus on considering procedure calls of multi-procedure programs, but also focus on achieving control flow representation simplicity and accuracy. In their CFG representation approach, Stafford et al. [45] provide inter-procedural control dependence
analysis to identify the dependencies among the statements of programs that are composed of pre-compiled libraries. Their approach consists of a control flow graph model and methodology for constructing the structure that comprises the graph. However, by focusing on the statement control dependence, their CFG structure belongs to the category of S-CFG which suffers from the same complexity symptoms mentioned earlier.

Sinha et al. [84] present an approach for computing inter-procedural control dependencies. They present a definition of inter-procedural control dependence that supports the relationship of control and data dependence to semantic dependence. In their work, control dependence is typically defined in terms of control flow graphs, paths in those graphs, and the post-dominance relations (among graph nodes). Different from our scope, their definition for control dependence involves the effects of predicate statements on the data and execution behavior of other statements. In our CDG, a control dependence relationship indicates only the execution flow structure between nodes. As a result, the CDG is more lightweight than their control dependence graph by considering simpler dependence relation among graph nodes. Therefore, it can be used by more research and practice techniques as underlying architecture representation. Moreover, in their work ([84]), the control dependence graph is derived from a CFG of a program by determining the control dependence among the statements. Since our dependence relation is defined based on the code structure, we are able to calculate the CDG directly from source code.

Aiming to solve the difficulty of the CFG size of large software programs, Balmas et al. [46] propose a decomposition method of a program into a hierarchy of groups where each group replaces a number of CFG nodes in the representation. This technique trades off between complexity and statement coverage. The nodes (or code statements) inside these groups are therefore, unseen and the control flow among them cannot be observed. The node grouping process could be an overhead and if applied differently,
it might lead to enormously different CFGs. In our approach, the use of procedure calls and event triggers as nodes decreases the size and the complexity of the control flow representation. Moreover, its meticulous graph structure does not lead to different control flow representations for a piece of code. Their decomposed CFG is analogous to C-CFG and consequently, it suffers from similar inaccuracy symptoms mentioned earlier.

2.2.2 Control Flow Monitor

Similar to error diagnosis, isolation, and recovery, error detection is an important aspect of fault tolerance technique to avoid system failures [85]. The expression “error detection” is sometimes used to indicate techniques for assuring system quality throughout the development phases including inspections, code reading, algorithm analysis, tracing, and control flow analysis [79]. However, these techniques belong to the fault avoidance and removal techniques for software reliability. Control flow analysis of these techniques aims to check the program control flow with respect to design problems (e.g., unreachable code or poor program structures). Numerous types of errors are addressed by the existing error detection techniques. Examples of these errors are overflow, underflow, divide by zero, memory allocation and deallocation, data flow, and control flow errors [86].

Control flow errors are major impairments to software system correctness. Operational environments of software systems are the main source of faults that can cause these control flow errors [62]. Software faults with respect to the operating system [59] and hardware faults with respect to the address circuits, program counter, and memory elements are the main causes of control flow errors [67]. Most of the hardware causes of control flow errors are related to the computing devices. Between 33% and 77% of transient hardware faults lead to software control flow errors [62, 65]. Recent industry trends in microprocessors aim to manufacture small size products with low supply voltage and high frequency. These trends are responsible for more susceptible transient hardware errors and control
flow errors in software systems operating on these microprocessors [68]. Moreover, some special system environments may have further sources of control flow errors. For example, different space radiations may cause control flow errors in a software system operating in the space environment [66].

Several approaches were proposed to detect [59, 62] and correct [64, 68] runtime errors among the interactions of the architectural elements (e.g., components, functions, object, process, task, state, etc.). Usually control flow error correction techniques use some form of redundancy to correct the runtime errors. The main goals of the current control flow error detection approaches are avoiding performance overhead [25, 59, 65], program modifications [27, 59], and the use of extra hardware [26, 61, 63]. The goals also involve increasing error coverage by considering both control flow and data flow errors [58, 68] or by considering intra-block and inter-block control flow transition errors [59–61, 67]. Some of these goals contradict each other. As a result, most of the existing techniques aim to trade-off between the former two goals on one side and the latter two on the other side.

Control flow error detection techniques mainly use signature-based approaches to monitor the control transitions among code instructions, system components, or basic blocks and detect their anomalies. Existing techniques detect control flow errors by the use of watchdog [59, 61], redundancy [26, 60, 64], assertion [27, 65], or monitoring approach [60, 61, 63]. Watchdog techniques aim to avoid the program modifications by using a watchdog processor. Michel et al. [59] describe an approach called WDP (Watchdog Direct Processing) which directly monitors the code instruction addresses of a main processor. By executing the same program operated by the main processor, the watchdog computes signatures of the executed instruction sequences and detects illegal execution paths. The major disadvantage of watchdog techniques is the need for additional hardware and the non-portability to various computing platforms [27].
2.2. RELATED WORK

Redundancy-based control flow error detection aims to detect and correct both control flow and data flow errors [87, 88]. Rebaudengo et al. [26] introduce data and code redundancy by performing a set of transformations on the high-level source code. Data flow errors are then detected by duplicating each variable and adding consistency checks after every read operation. Control flow errors are detected by duplicating the code instructions implementing each program operation and adding checks for verifying the consistency of the executed operations. Vemu et al. [64] present automatic correction of control flow errors using an algorithm involving addition of specific redundant code to the program. After assigning a unique Id for each program function, errors are detected by comparing the actual function Ids with the expected ones based on the program CFG. Errors are corrected by returning the control of the program back to the node that was being executed before the occurrence of a control flow error. Zarandi et al. [68] add redundant code instructions to a given program and detect the errors by comparing the actual control transitions and data transfers to the control and data flow graphs. Although these techniques go further beyond control flow error detection by addressing data flow errors and performing error correction, they involve a number of disadvantages. Redundancy is in general expensive and hard to obtain. By duplicating program variables and/or code instructions, the performance overhead will increase and system service will be degraded dramatically.

Assertion-based control flow error detection techniques aim to decrease the performance overhead and the use of extra hardware. They fortify a program with pre-inserted assertions in specific code locations. These assertions adjudge the control flow transitions among code instructions (or block) based on the injected signatures of each instruction (or block). Alkhalifa et al. [27] identify branch-free intervals (BFI) of code and insert assertions into the beginning and the end of these intervals. They detect the control
flow errors through modifications made on the GNU Compiler Collection (GCC). Vemu et al. [65] automatically embed extra code instructions into a program to continuously update run-time signatures and compare them against pre-assigned values. Although these techniques aim to decrease the performance overhead, the system performance is still degraded by the execution of the enormous number of assertions inserted into the program.

Signature-based control flow monitors aim to validate the control flow transitions in a separate software monitor. Thus, these techniques can save more performance overhead and at the same time they do not require any additional hardware or software redundancy. Signature-based monitoring approaches partition a software program into branch-free blocks [24,25,28,58,60–63,67]. Some of these techniques use finite state automata [24] or branch traces [62] to represent software architecture. However, the majority of them use the CFG for the same purpose [28,58,60,61,63,67]. The graph nodes are branch-free blocks and the graph edges are the control transitions among these blocks.

In general, signature-based monitors assign a unique signature for each basic block and inject it at the beginning and the end of the block to allow runtime monitoring of the program control transitions at its start and exit points. Control flow errors are detected by validating runtime signatures of the blocks against the program CFG. Aiming to increase the error coverage, Savaria et al. [60] present an error detection technique by considering intra-block transitions. By replicating program operations, the intra-block control flow is validated by using a check-pass flag between the original and the replicated operations. The inter-block control flow is checked by using dedicated global control variables describing the current program state. These variables allow the technique to predict the next set of instructions and to compare them with the actual transitions. Inserting a check-pass flag after each line of code inside a block could exaggerate the performance mitigation and overhead. Maghsoudloo et al. [58] present an automatic
2.2. RELATED WORK

control and data flow error detection and correction technique. Oh et al. [61] present a signature-based software monitoring technique for inter-block control flow. Sedaghat et al. [63] present a software-based error detection technique using encoded signatures. These techniques assign different arbitrary numbers or derive unique encoded signatures to the program basic blocks. Then, they follow the signature-based validation method mentioned earlier. The major limitation of these techniques is the use of branch-free block partitioning.

Due to complexity, some code statements neither fit into any basic block nor can they be partitioned into smaller blocks. Examples of these code complexities are control structures with procedure calls in their condition expressions and expression statements with multiple procedure calls. Connections that are invoked in these statements are ignored in the program partitioning and thus, they introduce CFG complexity through supplementary edges (see Subsection 2.1.3) to represent their invocations. Therefore, the ignored connections are becoming major factors in introducing illegal control transitions in the B-CFG representation. Consequently, in the existing techniques, validating the control flow transitions may be susceptible to false negatives due to the inclusion of these illegal control flow branches in the B-CFG. In comparison to the B-CFG, the CDG is a more accurate control flow representation that decreases the false negatives.

Few techniques have attempted to overcome the limitations of the B-CFG representation in the control flow error detection. Wu et al. [66] aim to avoid the decrease in error coverage due to the fan-in (multiple inward edges) nodes where several graph nodes are succeeded by a single node. In their error detection approach, they modify the CFG by eliminating these fan-in nodes. Li et al. [57] aim to decrease the performance overhead by avoiding the variations in the program block sizes and their effects on the injected monitoring instructions. They regroup the small basic blocks into larger sizes to achieve almost equal sizes of all basic blocks. Then, they apply a unified checking method for
the regrouped basic blocks represented using a modified CFG. Due to the simplicity of
the CDG, its use in control flow monitoring can decrease the performance overhead while
avoiding the aforementioned problem of the branch free block partitioning.

2.2.3 Failure Mode Prediction

Several research directions focus on fault and error predictions [89]. Faults and errors
are the sources and causes of system failures, respectively. Failure prediction is an effec-
tive reliability technique that was founded during the 1960’s decade to avoid the costly
maintenance and support forces of systems [11]. The technique has been used for hard-
ware systems (e.g., satellite [90] and cluster computing systems [91]) while the majority
of the recent approaches focus on failure prediction concerning software systems [3, 4, 92].
Contemporary autonomic and self-healing computing can depend on failure prediction
techniques to achieve resilient systems and avoid the possible failure hazards.

The common goals of the existing failure prediction techniques in software systems
involve forecasting the failure counts that may occur in future system runtime intervals,
the expected effects of failure modes, or the occurrence of failures during system runtime.
Predicting failure counts is a traditional reliability approach that predicts failure occur-
rence numbers in future time intervals [10, 11]. However, these approaches do not predict
failure occurrence at runtime nor they forecast any failure details (e.g., modes, effects, or
occurrence probabilities). Failure Modes and Effect Analysis (FMEA) techniques [12–17]
are utilized early in the development life cycle to enhance reliability through system de-
sign. These techniques forecast the potential failure modes and anticipate their severities
and consequent risks. Like failure count prediction, FMEA techniques also do not pre-
dict failures during runtime. Failure occurrence prediction techniques observe the system
behavior to forecast potential failures during system runtime [18–23]. In comparison to
these techniques, not only we predict runtime failures but also we anticipate their modes.
Failure occurrence prediction techniques forecast failures from anomalous execution behavior during system runtime. One of the major differences among these techniques is the execution parameters used to predict failures. Je et al. [19] predict failures from software inputs by using the maximum entropy principle. They forecast failures based on the statistical co-occurrence with inputs and the statistical cause-effect from faults. Fitzgerald et al. [14] predict failures using a machine learning technique by analyzing past system input features with both successful and unsuccessful outcomes. Predicting failures from inputs focuses only on operational faults and disregards the consequences of design faults on system errors. By depending on the details of system error log trails (lists of log records) to predict failures, our approach can consider failure causes of both design and operational faults.

A number of techniques focus on system execution states to predict runtime failures [20, 22, 23, 93]. These techniques select a number of system variables to express the system state and monitor their conformance with the software specifications to predict their related failures. Irrera et al. [93] propose an approach for selecting the most adequate software system variables for failure prediction. Hoffmann et al. [22] present a non-intrusive data-driven method for failure occurrence prediction as a continuous function of system variables. Aiming to decrease the difficulties of these techniques, Bai et al. [23] develop a stochastic model using Markov Bayesian Network for failure prediction by combining causal information and runtime observations in the estimation model. The disadvantages of these techniques are the need for design information and the extensive instrumentation of the monitoring instructions in software source code. For most software systems, log information can be available and exploiting them in failure prediction helps avoid the efforts of such instrumentation. Zhang et al. [20] predict runtime failures by constructing an on-the-fly model of near future system states and verifying them based on
a number of wanted/unwanted properties. The use of system error log to predict failures does not anticipate future system states. Rather, it simply estimates a predictive relationship of future failures based on the history of system execution error log and failure results.

Software system log data contains a wealth of information about a system execution state. Using these logs, potential failures can be predicted without any knowledge about software design and any need for software instrumentation. Cao et al. [18] utilize the failure history of a system to predict potential failures. They detect system states that are correlated with failures and predict their reoccurrence in future system runs. Salfner et al. [21] provide an error-based failure prediction by mining log information. They group similar error message sequences and use a statistical noise filtering approach based on these messages to predict future failures. In line with these techniques, we also use runtime history information to map system execution behavior with failure occurrences. However, we differ from these techniques by considering error parameter distribution among system logs to identify failure eventualities and predict their modes. Moreover, our method involves statistical analysis methods and thus it avoids the use of any stochastic relationships. Our approach is straightforward and does not require mining, identification, clustering, or filtering of system error logs to predict failures.

Our previous work on failure modes involves incorporating system’s multi-mode failure behavior into the reliability measures [31] and enhancing system reliability by trading off the criticality levels of different failure modes [94]. We also exploit the correlation between multi-mode failure flow behavior and architectural attributes [30] to provide a selection framework for incorporating reliability in the architectural design decisions to achieve software architectural solidarity [32].
2.3 Summary

The proposed techniques of this thesis facilitate failure prediction from the runtime errors of large software systems and include control flow representation, control flow monitor, and failure mode prediction. Three main CFG structures are distinguished in current reliability analysis work: Statement-based (S-CFG), Block-based (B-CFG), and Component-based (C-CFG) control flow graphs. These CFG structures are either remarkably complex or lack accuracy by allowing illegal control flow sequences in their representations. The proposed CDG contributes to the efficiency of the architecture-based reliability analysis by avoiding these complexities and inaccuracies. Moreover, unlike the existing CFG structures, the CDG is not limited to the inter-component control transitions and it preserves the connection invocation dependencies in the control flow representation.

Existing control flow error detection techniques detect errors by using watchdog, redundancy, assertion, or monitoring approaches. The major limitations in these techniques involve high performance overhead, intensive program modifications, and use of extra hardware. Our signature-based control flow monitor does not require any additional hardware or software redundancy. Existing signature-based control flow monitors are susceptible to false negatives due to the inclusion of some illegal sequences in the underlying control flow representation using B-CFG. The utilization of the CDG in our monitor decreases the possibility of false negatives due to its higher accuracy relative to the B-CFG. Moreover, our connection-based signature structure allows complete coverage of all program connections, and it can determine the control flow errors with respect to the unreturned connections in case of improper program terminations.

Most existing failure prediction techniques do not forecast failures during system runtime. Instead, they forecast failure numbers and estimate the effects of possible failure
modes that may occur in future time intervals. Existing runtime failure prediction tech-
niques do not estimate the failure modes and do not predict the eventuality of failure
occurrence. Our approach is able to predict the failure eventuality during runtime and
forecast the possible failure modes.
Chapter 3

A Control Flow Representation Technique

Control Flow Graph (CFG) is an essential tool for software engineering research and practices. In this chapter, we propose a Connection Dependence Graph (CDG) representation for software control flow that avoids the complexity and inaccuracy of the existing CFG representations. A CDG is a connection-based control flow graph representation of possible connection sequences that might be traversed through a program during its execution. We describe the CDG and explain how to derive it from program source code.

We first provide a reverse engineering technique for capturing a software architecture from program source code (Section 3.5). Then, we define a number of architectural and transitional patterns and use them to derive the CDG. Briefly, an architectural pattern includes some statement types with similar control branching structures and a transitional pattern indicates a group of transitions with similar structures. The transitional patterns provide grouping of the possible transitions in different architectural patterns. We exploit these architectural and transitional patterns to capture possible control transitions of each code statement categorized into a number of groups that are specified based on these transitional patterns. The captured transition groups are then used to build the program CDG (Section 3.6). We provide a case study to examine the effect of program size on the CDG and the other CFGs by comparing them with respect to complexity and inaccuracy using the PostgreSQL open source database system [95] (Section 3.7). In
3.1 Overview

Figure 3.1 provides an overview of the proposed technique. The input of our approach is a program source code and the output is a program CDG. From the program source code, we obtain the software architecture by using our reverse architecting toolkit which derives the software architectures of C-based programs. Next, from the software architecture, we derive the transition groups of each program statement. In this derivation, we utilize the specification of statement structures (architectural patterns) and transitional patterns to generate a number of transition groups for each statement. Finally, we provide an algorithm that uses these transition groups to build the program CDG.

3.2 An Illustrative Example

We provide a code example to help describe the different CFGs and use it as a running example throughout the thesis. The code example shown in Figure 3.2 is written in C language and it implements the Depth First Search\(^1\) (DFS) algorithm. By considering

---

\(^1\)Depth First Search (DFS) is an algorithm for traversing a tree structure that explores the first child of a each node before backtracking to its siblings.
3.2. AN ILLUSTRATIVE EXAMPLE

Figure 3.2: An example C code for Depth First Search (DFS) algorithm.

```c
/*File 1: main file */
int main(){
    dfs(readAdjMat(), 0);
    return 0;
}
/*File 2: DFS file */
int stacktop= -1;
int stack[5];
int visited[],i;
int[5][5] readAdjMat(){
    return(((0,0,1,1,0),(0,0,0,1,0),(0,1,0,1,1),(0,0,0,1,0),(0,0,0,1,0)));
}
int dfs(int adj[][],int root){
    attachNode(root);
    while(peek() != -1){
        int start= peek();
        for(i=0; i<5; i++){
            if(adj[start][i] && visited[i] == 0){
                attachNode(i);
                break;
            }
        }
        if(i==5){
            stacktop--;
        }
    }
    return 0;
}
int peek(){
    return stack[stacktop];
}
void attachNode(node, int visited[]){
    stack[stacktop++] = node;
    visited[peek()] = 1;
    printf("%d-", node);
}
```

Each code file as a separate component, this example includes two components $c_1$ and $c_2$ that correspond to File 1 (main file: Lines 1-5) and File 2 (DFS file: Lines 6-36), respectively. Component $c_1$ includes one procedure (main()) and component $c_2$ includes 4 procedures (readAdjMat(), dfs(), peek(), and attachNode()) which are called from several code locations in this code example.
Figures 3.3(a), 3.3(b), and 3.3(c) show the S-CFG, B-CFG, and C-CFG of the provided code example respectively. Two virtual nodes S and E represent the start and the end of each CFG, respectively. In Figure 3.3(a), each node is named over the corresponding code statement indicated by its line number. At each node, directed edges indicate control flow transitions to or from other code statements. In Figure 3.3(b), graph nodes are branch-free blocks and are named after the range of included lines. Depending on the inter-procedure
3.3 Complexity and Inaccuracy of the Existing CFGs

Some of the limitations of the existing CFG structures (S-CFG, B-CFG, and C-CFG) are discussed in the previous chapter. A major difference among the existing CFGs is in the representation of nodes. A node representation affects the number of nodes in a graph and the number of edges among them. Relatively small size nodes (e.g., statements in the S-CFG) allow comprehensive control flow representations and increase the number of graph edges. Conversely, large size nodes (e.g., components in the C-CFG) are less meticulous with respect to the dependencies among control flow transitions and uninformative about the control flow within these nodes. However, based on the component representation (e.g., function, module, file, class, or any other component forms), C-CFGs may vary with respect to complexity and accuracy. In this thesis, we consider that, a component represents a program file.

To proceed with our discussion, we categorize the edge types of a CFG into three main types: interface, loop, and transitional edges. Interface edges are used to represent connection invocations and returns. Loop edges are used to represent iterative transitions inside loop blocks. Transitional edges are used to represent the control transfers according
to the control structure code instructions. Obviously, a CFG node representation affects the number of edges of these three types. However, in this section, we only focus on the interface edges as a sample edge type with respect to its effect on the complexity and accuracy of the existing CFG structures.

Connection invocation and return (interface) edges of a CFG depend mainly on the Connection Implementation Block (CIB)\(^2\) structures and invoker locations of these connections. Therefore, we discuss the effects of interface edges on a CFG based on the representation of the block start, exit, and return transitions. Figure 3.4 provides illustrative examples of these edges. These start, exit, and return edges represent control transfers to (or from) virtual nodes $S$ (start) and $E$ (end). As shown in the figure, a start edge connects an incoming connection to a start node of a code block. An exit edge connects an exit node of a code block to a connection return. A return edge connects a returning connection to a subsequent statement of a connector. In the following paragraphs, we provide some examples for the edge types from the running example of Figure 3.2.

Figure 3.4: Types of Control Flow Graph (CFG) edge representations.

Let $\text{tran}(i, j)$ indicate a control flow transition from node $i$ to node $j$. Let us also refer to

\(^2\)A CIB of a connector is the code block representing its connection implementer.
a CIB as \( cib(c, b) \), where \( c \) is the component that includes the CIB and \( b \) is the CIB Id or name. We indicate the start and the end of a block as \textit{start} and \textit{end} respectively. Using the same notation, we classify \( tran(start, 14) \) in the code example as a block start edge of the CIB of procedure \( dfs (cib(2, dfs)) \). \( tran(start, 33) \) is start edge of \( cib(2, attachNode) \). \( tran(27, end) \) is an exit edge of \( cib(2, dfs) \). As example return edges, we consider the procedure \textit{peek} in the running example. This procedure is called in statements 15, 16, and 34. The return edges of \textit{peek()} call in Line 15 are \( tran(15, 16) \) and \( tran(15, 27) \).

In the following subsections, we discuss the complexity and inaccuracy aspects of these CFGs due to connection invocations and returns, supplementary edges, and dependence relationships.

### 3.3.1 Complexity

Since the C-CFG disregards the intra-component control transitions, the start, exit, and return of a CIB does not increase or decrease its complexity. In the following paragraphs, we discuss the complexity effects of these edges on the S-CFG and B-CFG. We denote the number of start, exit, and return edges of a block \( b \) as \( b_{\text{start}} \), \( b_{\text{exit}} \), and \( b_{\text{return}} \), respectively. Let \( r \) be a connector in a block \( b \) and consider that it executes a code block \( b' \). In the following paragraphs, we use these notations to express the complexity impacts of connection invocations and returns denoted as \( r_{\text{invoke}} \) and \( r_{\text{return}} \), respectively. Connection invocations do not introduce any extra branches with respect to S-CFG. In this CFG, every procedure implementation block (procedure body) has one start node that represents the procedure declaration statement. Therefore, \( \forall r: r_{\text{invoke}} = b_{\text{start}} = 1 \). However, in B-CFG, due to the avoidance of compound statements of the selected branch-free blocks, a procedure declaration statement is not represented by any nodes in the graph. Consequently, a connection invocation complicates a B-CFG by introducing a number of start edges from a caller node to every possible start node of its target code block. Therefore, in
B-CFG, $\forall r: r_{\text{invoke}} = b_{\text{start}}; b_{\text{start}} \geq 1$, i.e., any connection invocation is represented using one or more edges depending on the number of start edges of the invoked code block.

Connection returns are a major complexity factor with respect to both S-CFG and B-CFG. To represent a connection return, an edge from every possible exit node of the callee block to every subsequent node of the caller node is included in the graph. Therefore, in both S-CFG and B-CFG, $\forall r: r_{\text{return}} = b_{\text{exit}} \times b_{\text{return}}; b_{\text{exit}} \geq 1$ and $b_{\text{return}} \geq 1$. The major implications of these graph complexities include state-explosion, extra storage and processing requirements, and long execution traces with respect to their derivations and manipulations.

### 3.3.2 Inaccuracy

Due to the complexity of the S-CFG and the B-CFG and the high abstraction of the C-CFG, a large number of illegal control flow sequences may be derived by each of these graphs. Two major factors are responsible for illegal control flow sequences in both S-CFG and B-CFG: control return edges and supplementary edges. Likewise, in C-CFG, lack of connection dependence representation is the main cause of these illegal sequences. *Connection dependence* indicates a specific order of execution with respect to program connections due to code structure and programming language execution priorities. In the following paragraphs, we discuss the impacts of these factors on the inaccuracy of the CFG structures.

In addition to its effect on the S-CFG and B-CFG complexities, control returns are a major source of illegal control flow sequences in these graphs. In these CFGs, if a procedure is called more than once from different code locations, then, as shown earlier, a number of edges ($r_{i\text{return}}$) is required to represent these calls from the exit nodes of a callee block towards every subsequent node of each different call $i$. Given that, a control
flow sequence according to these graph representations may involve an edge representing a procedure call followed by a return edge to a different call of the same procedure in another code location. The number of return edges from a specific callee block to a number of calls $I$ is calculated as $\sum_{i=1}^{I} r_{i}^{\text{return}}$. Therefore, we can derive the number of possible illegal branches due to a connection return in S-CFG and B-CFG denoted by $\lambda_{j}^{\text{return}}$ as $\sum_{i=1}^{I} r_{i}^{\text{return}} : i \neq j$, where $r_{j}$ is one of these calls. Let us represent a control flow sequence as $\text{seq}(i, j, \ldots, k)$, where $i, j$, and $k$ represent node Ids. Each two consecutive nodes (e.g., $i$ and $j$) are connected using $\text{tran}(i, j)$. Using this notation, we provide some example control flow sequences based on the running example of Figure 3.2. These sequences are $\text{seq}(15, 30, 17)$, $\text{seq}(15, 30, 18)$, $\text{seq}(15, 30, 23)$, and $\text{seq}(16, 30, 35)$. These example illegal sequences are already presented in the S-CFG (see Figure 3.2).

Supplementary edges are introduced to a CFG structure to represent the ignored connections, i.e., connections that do not belong to any graph nodes according to the graph structure. An example of these ignored connections in the S-CFG is the multiconnection expression statement. An example of these ignored connections in the B-CFG is the language control structure statements (constructs) with connections in their condition expressions. In the following two paragraphs, we describe the effects of these ignored connections on the S-CFG and B-CFGs.

In the S-CFG, each program statement is represented by one graph node. Very often, several connections may exist in one expression statement (e.g., Line 3 of Figure 3.2). In this case, the supplementary edges are introduced to the graph to represent the control transfer among program CIBs according to the execution sequence of the invoked connections in this statement based on the programming language execution priorities. As explained in Subsection 2.1.3, these supplementary edges connect the end of the first invoked CIB to the start of the next invoked CIB. These edges then become possible sources
of illegal control flow sequences. If \( r_i \) and \( r_j \) are two connections in a single statement and are executed in order, then we express the number of incorrect transitions due to the return of connection \( r_i \) in this statement as 
\[
\lambda^\text{return}_i = \sum_{k=1}^{K} b_{k}^{\text{start}} \times b_{i}^{\text{exit}} \times b_{j}^{\text{start}} : k \neq i.
\]
For example, to represent the two procedure calls in Line 3 of Figure 3.2, an edge \( \text{tran}(11, 13) \) will represent the return from \( \text{readAdjMat}() \) to the \( \text{dfs}() \) procedure. Depending on the number of invocations to the \( \text{cib}(1, \text{readAdjMat}) \), a number of illegal sequences can be introduced due to the existence of the \( \text{tran}(11, 13) \) edge in the control flow representation.

The B-CFG is similarly affected by this type of ignored connections, since a basic block partitioning does not represent the program statements with more than one connection in the graph nodes and consequently, it involves supplementary edges to represent their transitions.

The B-CFG involves also another type of ignored connections which belong to a control structure condition expression (e.g., Line 15 of Figure 3.2). These connections are not associated with any block since they exist in the start of a compound statement that may have several other branches. Thus to represent the invocations of this type of ignored connections, a number of edges from every possible block that precedes it \( (r_i^{\text{ancestor}}) \) to every possible start block \( (b_{\text{start}}) \) of the callee procedure must be supplemented to the B-CFG. To represent the return of the ignored connection \( (r_i) \), a number of edges from every possible exit block \( (b_{\text{exit}}) \) of the callee procedure to every possible block that succeeds the connector statement \( (r_i^{\text{successor}}) \) must also be supplemented. These edges also become possible sources of illegal control flow sequences. If \( r \) is an ignored connection in a B-CFG representation, then we express the number of incorrect transitions due to the return of connection \( r \) as 
\[
\lambda^\text{return}_i = r_i^{\text{ancestor}} \times b_{\text{start}} \times \sum_{j=1}^{J} r_j^{\text{return}} + \sum_{k=1}^{K} r_k^{\text{invoke}} \times b_{\text{exit}} \times r_i^{\text{successor}} : i \neq j \text{ and } i \neq k.
\]
For example, to represent the procedure call in Line 15 of Figure 3.2 in a B-CFG representation, the supplementary edge \( \text{tran}(14, 30) \) will represent the invocation of the procedure \( \text{peek()} \) in Line 15 and two supplementary edges \( \text{tran}(30, 16) \) and \( \text{tran}(30, 27) \)
3.3. COMPLEXITY AND INACCURACY OF THE EXISTING CFGS

will represent the procedure return to the antecedent blocks of Line 15. Depending on the number of invocations to the cib(1, peek), a number of illegal sequences can be introduced due to the existence of these supplementary edges in the control flow representation.

In C-CFG, edges from one node indicate inter-component interface connections. Connections of a software component are dependent on each other according to the internal code of this component. Unfortunately, C-CFG does not provide enough information about the dependence relationships among the connections that are invoked in a single component. Therefore, C-CFG considers all outward edges of a specific node as possible transitions at all visits\(^3\) to this node and disregards their dependencies. A number of visits to a component can take place in a single system run. As a result, any C-CFG node that has dependent outward edges to other nodes is considered a possible source of illegal control flow sequences. The number of illegal control flow sequences in this graph increases proportionally with the number of dependent outward edges of each node. It also increases exponentially with the number of nodes that have dependent outward edges and that are connected using any execution path. If \(d_c\) is the number of dependent outward connectors of a component \(c\), then the number of possible incorrect transitions at each connection \((r)\) invocation of this component is \(\lambda_r^{\text{return}} = (d_c - 1) \times (r^{\text{ancestor}} + r^{\text{successor}})\). Although C-CFG is the simplest graph among the existing structures, it has a serious problem with respect to control flow sequence validity. As example of these illegal sequences from Figure 3.2, we consider the outward connections of component 2 (DFS file) as shown in Figure 3.3(c). At any system state where component 2 is executed, the figure displays 6 possible control flow transitions. Therefore, according to the C-CFG, \(\text{seq}(14, 33, 30)\), \(\text{seq}(14, 33, 33)\), \(\text{seq}(19, 33, 30)\), \(\text{seq}(16, 30, 33)\), and \(\text{seq}(34, 30, 30)\) are possible sequences. In fact, these sequences are not legal since they disregard the system states during the component visits.

\(^3\)A visit to a component is an interval of time in which the component receives the control, executes a block of code, and then returns or ends.
3.4 Connection Dependence Graph

Connection Dependence Graph (CDG) represents software control flow based on the dependencies among system connections. Unlike the existing CFGs in current reliability analysis, it provides a simple representation of software control flow with lower number of illegal control flow sequences.

Figure 3.5: A CDG graph representing the example code of Figure 3.2.

A CDG is a directed connected cyclic graph $D = (N, E)$, where $N$ is a set of nodes and $E$ is a set of directed edges. Each node of the graph is a connector (procedure call, event trigger, or class method). Each edge in the graph indicates a dependence relation between two connections by specifying a deterministic execution order of their connectors. The execution orders of these connectors are determined based on the control flow structures and the call sequences of a program. Two virtual nodes are attached to a CDG indicating the start and the end of the execution. Figure 3.5 shows the CDG of the code example of Figure 3.2. In Figure 3.5, each node is named over the corresponding code statement indicated by its line number. At each node, directed edges indicate control flow transitions to or from other connectors.

Although, the CDG ignores code statements that do not involve any connectors, these statements are less likely to cause control flow errors and are less interesting for control flow monitoring of component-based software systems. Likewise, representing these statements in separate nodes may introduce undesired complexity to the CFG representation and may increase the state space of a program execution as in the case of S-CFG.
3.5. REVERSE ENGINEERING FOR SOFTWARE ARCHITECTURE

Deriving a CDG from source code entails a preparatory syntax analysis to determine the grammatical structure of the source code with respect to a given formal grammar. In the following two sections, we derive a program CDG in two steps. First, we derive the software architecture from program source code (Section 3.5), second, we utilize this software architecture to derive the program CDG.

3.5 Reverse Engineering for Software Architecture

Most existing software reverse architecting techniques and tools focus on the graphical rather than the data representation of software architectures [96, 97]. Data representations of software architectures allow automatic incorporation of these representations in software research and practices. In the following subsections, we describe our reverse architecting approach and toolkit for capturing software architectures from source code and representing them using Extensible Markup Language (XML) [98]. In the following subsections, we first provide an algebraic definition of software architecture. Next, we present a static analysis approach for extracting basic architectural information from the source code and explain how to aggregate this information to obtain the software architecture. Then, we describe our data representation method using XML schemes. The three steps of our technique (extraction, aggregation, and representation) focus on the C-based programs and consider C-files as system components. Procedure calls and event triggers are captured as connections among these components.

3.5.1 Algebraic Definition of Software Architecture

Several software engineering techniques represent a program structure algebraically in n-tuple forms [99–101]. Similarly, we define a software architecture $A$ as $A = (C, B, L, S, R)$, where $C$ is a finite set of components which can represent program files, modules, classes,

---

4“Syntax tree” and “program structure” are sometimes used interchangeably in this thesis.
model-based components, etc. Each component is partitioned into a number of blocks in $B$. Each block represents a CIB in this component. Therefore, a component $c_i \equiv \{b_j: j = 0, 1, \ldots, J\}$, where $J$ is the number of blocks in component $i$ and $\forall_{c_i} \\forall_{b_j \in c_i} \ b_j \in B$. Each block $b_j$ is associated with a statement list $l_j$ in $L$, where $l_j = [s_0, s_1, s_2, \ldots, s_K]$ and $K$ is the number of statements in the list $l_j$. Therefore, $\forall_{b_i} \ \forall_{s_k \in b_i} \ s_k \in S$. Depending on its control structure, a statement can recursively be associated to a number of nested statement lists in $L$. Finally, each statement can be associated to zero, one or more connectors in $R$. A connector $r$ in $R$ can be associated to only one statement in $S$. The set of connectors $R$ includes all the event triggers and procedure calls of a software program.

Table 3.1 describes the software architecture representation of the code example of Figure 3.2 based on the aforementioned tuple form. The table rows are the code lines in this example and the columns are the elements of the representation: component, CIB, parent code list, statement (or sub-statement) type, statement sub-list(s), and the connectors of these statements. Some code lines are not associated with any CIB since these statements are comments (e.g., Line 1), punctuator (e.g., Line 5), block declarations (e.g., Line 2), or global declarations (e.g., Line 7). To explain the table contents, we refer to Line 13 of the running example. Line 13, provides a declaration of a CIB for the $dfs()$ procedure. Columns 1 and 2 show the line number and component Id, respectively.

Since Line 13 is a procedure declaration statement then, column 3 identifies the $cib(2, dfs)$ of this procedure declaration statement as block $b_3$. Column 4 shows the parent code list of a statement if it belongs to any. We consider that, the CIBs are the roots of their control trees and thus we ignore their parent code list, since it is always trivial (the containing file). Given that, column 4 of Line 13 is empty. Column 5 describes the statement (or sub-statement) type. Column 6 presents the code list(s) of Line 13 as $l_3$. Finally, column 6 lists the connectors that exist in Line 13 (none in this case). From the table, $l_3$ appears three times as parent list in column 3. This means that, $l_3$ includes
### Table 3.1: The software architecture representation of the code example of Figure 3.2.

<table>
<thead>
<tr>
<th>Line</th>
<th>Comp.</th>
<th>CIB</th>
<th>List</th>
<th>Arch.Pattern</th>
<th>Sublists</th>
<th>Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Comment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$b_1$: main</td>
<td>$l_1$</td>
<td>Declaration</td>
<td></td>
<td>$l_1$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>main</td>
<td>$l_1$</td>
<td>Expression</td>
<td></td>
<td>$r_1$: readAdjMat, $r_2$: dfs</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>main</td>
<td>$l_1$</td>
<td>Jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Comment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>Declaration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>Declaration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>Declaration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>$b_2$: readAdjMat</td>
<td>$l_2$</td>
<td>Declaration</td>
<td></td>
<td>$l_2$</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>readAdjMat</td>
<td>$l_2$</td>
<td>Jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>$b_3$: dfs</td>
<td>$l_3$</td>
<td>Declaration</td>
<td></td>
<td>$l_3$</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>dfs</td>
<td>$l_3$</td>
<td>Expression</td>
<td></td>
<td>$r_3$: attachNode</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>dfs</td>
<td>$l_3$</td>
<td>Loop</td>
<td>$l_4$</td>
<td>$r_4$: peek</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>dfs</td>
<td>$l_4$</td>
<td>Expression</td>
<td></td>
<td>$r_5$: peek</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>dfs</td>
<td>$l_4$</td>
<td>Loop</td>
<td>$l_5$</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>dfs</td>
<td>$l_5$</td>
<td>Condition</td>
<td>$l_6$</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>dfs</td>
<td>$l_6$</td>
<td>Expression</td>
<td></td>
<td>$r_6$: attachNode</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>dfs</td>
<td>$l_6$</td>
<td>Jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>dfs</td>
<td>$l_4$</td>
<td>Condition</td>
<td>$l_7$</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>dfs</td>
<td>$l_7$</td>
<td>Expression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>dfs</td>
<td>$l_3$</td>
<td>Jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>$b_4$: peek</td>
<td>$l_8$</td>
<td>Declaration</td>
<td></td>
<td>$l_8$</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>peek</td>
<td>$l_8$</td>
<td>Jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>$b_5$: attachNode</td>
<td>$l_9$</td>
<td>Declaration</td>
<td></td>
<td>$l_9$</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>attachNode</td>
<td>$l_9$</td>
<td>Expression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>attachNode</td>
<td>$l_9$</td>
<td>Expression</td>
<td></td>
<td>$r_7$: peek</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>attachNode</td>
<td>$l_9$</td>
<td>Expression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>2</td>
<td>Structure end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3 statements (14, 15, and 27). Statement 15 is a loop and it has a sub-list $l_4$ where $l_4$ includes other two nested statements 16 and 17. Some statements include connectors (e.g., statement 14 has a connector `attachNode()` referred as $r_3$).
3.5.2 Extracting Basic Architectural Information

We describe our method for extracting software architecture from source code. In this extraction step, we construct the program code statements from the source code in a basic data structure and obtain some basic information about these statements. The input of this step is a program source code and the output is a tree data structure representing the statements and their basic information. We start this step by scanning the project folder structure to preliminarily identify the project files (C-files: “.c”, and header files: “.h”) and to generate lists of pointers to these files. These lists can also be updated by adding or removing other files to the project, if they are explored in later stages. After building the list of files, we provide three major processes for all the program files: lexical analysis, preprocessing, and syntactic analysis. Starting from the main file (usually called ”main.c”) file and throughout these processes, we keep updating the preliminary list of project files. Therefore, at the end of this stage, we obtain the accurate project structure. In the following paragraphs, we describe the lexical analysis, preprocessing, and syntactic analysis methods that are implemented in our approach.

Lexical analysis

The lexical analysis process converts raw text buffers into sequences of tokens and identifies their attributes. A token is a string of characters structured according to a lexical grammar [102] of a programming language. During this process, the text buffer is continuously scanned and tokens are preliminarily identified in a number of stream types: word, special-character, string-literal, comment, header file name, constant, char constant. After a complete capture of a token, the token type is identified from the stream type based on the lexical grammar. Afterwards, a new stream type starts and its type is again identified during the capturing process. With each captured token, we store some of its attributes including token type (e.g., keyword, identifier, operator, punctuator, and
Preprocessing

Sequences of tokens of the program files are now ready for the second process in this step: the preprocessing. This process reproduces the token sequences of the program after applying textual substitutions and macro expansions based on the preprocessing directives existing in these token sequences. In the C-language, the preprocessing directives are identified in the token sequences by the \# character which always starts these directives (e.g., \#if, \#ifdef, \#ifndef, \#elif, \#else, \#endif, \#undef, and \#define). Program files with \#include directives in their token sequences are preprocessed after preprocessing all its included header files. However, if a header file itself has other \#include directives, it is then preprocessed after all these header files. The preprocessing is a recursive process where a macro can be expanded using a code segment that itself includes other macros that again needs to be expanded. After preprocessing all the program files, the newly reproduced files are then ready for the syntactic analysis process.

Syntactic analysis

The previous two processes (lexical analysis and preprocessing) generally deal with individual tokens and do not provide any knowledge about the token combinations. The third process (the syntactic analysis) determines the grammatical structure of the token sequences with respect to the C-language grammar [102]. Our syntactic analysis process follows a recursive descent parser methodology. A recursive descent parser is a top-down parser that is built from a set of mutually recursive procedures where each procedure implements a production rule of a language grammar [103]. Similar to preprocessing, program files with \#include directives in their token sequences are parsed only after parsing all its included header files.
The syntactic analyzer passes each C-file to a parsing procedure called *read-block* that captures the syntax tree of its token sequence. The *read-block* procedure reads individual tokens and constructs the block statements one at a time by calling *read-statement* procedure. The *read-statement* detects the statement start and end, identifies its type, reads the statement parts, and records some information about the contained identifiers in this statement. This procedure first, detects the statement end token location by matching left and right braces and brackets and/or checking the semicolons and end of lines. The *read-statement* procedure then identifies the statement type by glimpsing the next few tokens. The currently parsed statement is preliminarily identified as declaration, selection, iteration, expression, or jump statement. Based on this preliminarily identification, the parser proceeds with other procedures (*e.g.*, *read-declaration-statement* and *read-selection-statement*). Later during this process, each of these procedures calls other procedures for lower level grammar rules. For example, the *read-selection-statement* procedure further detects the statement type whether it is an *if* or a *switch* statement and consequently calls *read-if-statement* or *read-switch-statement*, respectively. When any control structure statement is detected, the parser reads the statement header and again passes the statement block(s) recursively to the *read-block* procedure by specifying other (narrower) block start and end locations. This process recursively continues until the whole file is parsed, then it continues through the other program files. After a statement is read, all the identifiers in this statement are associated with identifier types. The type of each identifier is recorded in its token attributes and these identifiers are then considered to read the next statements and identify their types within the scopes of these identifiers. Examples of the identifier types are *typedef*, *enum-type*, *enum-constant*, *struct-type*, *union-type*, *Union-Element*, and *Variable*. 
3.5. REVERSE ENGINEERING FOR SOFTWARE ARCHITECTURE

3.5.3 Aggregating Architectural Information

This step mainly organizes the captured software architecture in an intermediate data structure to prepare for exporting it into XML documents. This step includes two main processes: tree generation and node identification. In the first process, we build a data tree for each C-file of the program to represent its syntax tree. Figure 3.6 illustrates the intermediate tree data structure for representing the captured software architecture of the program files. Each file represents a system component. Each component is associated with a tree data structure indicated as thick arrows. The root of each tree is a main code block $b_i$ that represents the whole file, where $i$ is the file (or component) Id. The successor nodes of each block represent (in order) the list of statements it contains: $s_1, s_2, \cdots, s_m, \cdots, s_M$. In this level of the tree, statements are procedure declarations, global variable declarations, or preprocessing directives. If a statement $s_m$ contains any code blocks (e.g., procedure declaration), then these blocks are represented in the data structure by successor nodes and they represent the component CIBs. These successor nodes

![Diagram of intermediate tree data structure]

Figure 3.6: Intermediate tree data structure for representing a software architecture.
are again blocks that recursively may contain other statements. With each statement and block, we store some basic information. Example of this information includes the code location, the parent block, child blocks (if any exists), statement type, token list, included variables, and read/write operations.

Further information involve the call invocations in each statement and the call parameters whether they are passed by values, by references, or by another value-returning procedure calls (functions). In the second process of this step, we provide a unique Id for each block, statement, procedure call. In C-language, the file structures allow some types of statements to exist in C-files and others to exist in header files. The code statements in a C-file are usually preprocessing directives or declaration statements for variable, types, procedures, and others. Therefore, in our tree data structure, the tree root (e.g., $b_1$, $b_2$, and $b_n$, indicated as squares in the figure) represents the whole file. The next level in the tree (e.g., $s_1$, $s_2$, and $s_m$, indicated as circles in the figure) represents the declaration statements in this C-file. If these statements are procedure declaration, then the third level represents the CIBs of the implemented procedures. The fourth level and up represent the code structure inside each of these CIBs.

3.5.4 Representing Software Architecture

In our representation of software architecture, we use Extensible Markup Language (XML) to represent the extracted software architecture. XML is a markup language that allows data storage in document-oriented databases. It defines a set of rules for encoding self-descriptive data structures in a human and machine readable format. The XML documents are platform independent and they can be deployed and utilized by software engineering practices in different software system enterprises and computing environments. In this step, we structure two types of XML documents and generate the data representation
of the extracted software architecture based on these structures. Two types of XML documents are generated: system structure and component structure. The system structure XML document provides basic information about the project structure and components. For each component, this document lists the number of CIBs, folder location, and the location of the XML document presenting its structure (see the example in Figure A.1).

A component structure XML document is generated for each program component. It provides identification information of the component and describes its structure in a tree format. Each CIB is associated with a code block (referred as “Stat_block” in the XML document). A code block may include a list of statements. A statement can be selection, condition, loop, expression, or jump. Depending on the statement type, a statement can recursively include other code blocks. These code blocks are expanded to list their contents. Each expression statement is associated with a sub-tree displaying the variable accesses and the call invocations in this statement. A call invocation can recursively include other expressions as call parameters, which again can include other variable accesses and call invocations. For each CIB in the component, the XML document lists the inward (or provided) and outward (or required) connections of that CIB. These connections are listed by their connection Id. The listed connections include procedure calls, event triggers, and data access read/write operations (see the example in Figure A.2). The XML documents are generated from the intermediate tree data structure of the previous step. The tree is scanned and XML parts are written into the documents by using recursive procedures for each block, statement, and connection. In Figure 3.7, we display the XML representation of the example code of Figure 3.2. Three XML files are displayed: the system structure document and two component structure documents.
3.5. REVERSE ENGINEERING FOR SOFTWARE ARCHITECTURE

(a) System structure

```xml
<?xml version="1.0"?>
<DOCTYPE Component>
  <Component Num_CIBs="1" Name="main.c" Id="1">
    + <CIB Name="Master" Id="0">
      + <CIB Name="main" Id="1">
        + <Provide Num_variable_access_times="0" Num_call_invokes="0">
          + <Require Num_variable_access_times="0" Num_call_invokes="2">
            - <Stat_block File_location="(2,11),(5,2)" Braced="Yes" Num_statements="2">
              - <Expression File_location="(3,5),(3,35)" Data_Access_statement_type="Expression">
                - <Call Name="dfs" Id="1" Home_CIB="(2,1)" Access_CIB="(1,1)" Collect="" />
                <Call>
                </Call>
              </Expression>
            </Stat_block>
          </Require>
        </Provide>
      </CIB>
    </CIB>
  </Component>
</DOCTYPE Component>
```

(b) Structure of component 1

```xml
<DOCTYPE Component>
  <Component Num_CIBs="4" Name="dfs.c" Id="1">
    + <CIB Name="Master" Id="0">
      + <CIB Name="dfs" Id="2">
        + <Provide Num_variable_access_times="0" Num_call_invokes="1">
          + <Require Num_variable_access_times="0" Num_call_invokes="4">
            - <Stat_block File_location="(13,1),(28,2)" Braced="Yes" Num_statements="3">
              - <Expression File_location="(14,5),(14,22)" Data_Access_statement_type="Expression">
                - <Loop Id="15" File_location="(15,5),(26,6)">
                  - <Stat_block File_location="(15,22),(28,2)" Braced="Yes" Num_statements="3">
                    - <Call Name="peak" Id="3" Home_CIB="(2,3)" Access_CIB="(2,2)" Collect="" />
                      <Call>
                        + <Expression File_location="(16,9),(16,26)" Data_Access_statement_type="Call">
                          + <Loop Id="15" File_location="(17,9),(22,10)">
                            + <Condition File_location="(23,9),(25,10)"
                              </Condition>
                          </Loop>
                        </Expression>
                      </Call>
                    </Stat_block>
                  </Stat_block>
                </Loop>
              </Expression>
            </Stat_block>
          </Require>
        </Provide>
      </CIB>
    </CIB>
  </Component>
</DOCTYPE Component>
```

(c) Structure of component 2

```xml
<DOCTYPE Component>
  <Component Num_CIBs="4" Name="dfs.c" Id="1">
    + <CIB Name="Master" Id="0">
      + <CIB Name="attachNode" Id="4">
        + <CIB Name="peak" Id="3">
          + <CIB Name="attachNode" Id="4">
        </CIB>
      </CIB>
    </CIB>
  </Component>
</DOCTYPE Component>
```

Figure 3.7: The XML representation of the example code of Figure 3.2.
3.5.5 Toolkit Description and Limitations

We developed a toolkit based on the provided approach and used it throughout our experimental evaluations in this thesis. The toolkit is able to capture the architecture of ANSI-standard C-based programs. The toolkit is built using Java 2 Platform Enterprise Edition (J2EE) [104]. The toolkit object oriented program includes 39 main classes and it is compiled on Windows 7 platform. In addition to extracting the architectural information of C-based programs, this toolkit also performs code instrumentation and fault injection. Our toolkit can capture the architecture of large C-based software programs. It does not need any instrumentation in the GNU makefiles and it is not GCC-based\(^5\). Moreover, it is a command line tool that needs only few parameters to start: the project folder, the main.c file location, and an output XML folder. These advantages make our tool more suitable to be used by researchers and practitioners in the software engineering field for extracting the architecture of C-based programs.

The tool includes a few limitations as follows. During the preprocessing of a C-file, the toolkit considers all the preprocessing directives in this C-file and in all of its header files. However, if these header files include other header files using the \texttt{#include} directive, they are not considered in the preprocessing of the original C-files. In large software programs, this may result in unknown identifiers and consequently, some statements may be unrecognized. However, the tool marks these statements as “unknown” and avoids their representation in the XML documents. Another limitation is related to the \texttt{#if} directives. The condition part of these directives is not completely evaluated in our tool. The tool only checks for simple conditions like \texttt{#ifdef} or \texttt{#ifndef}. This may cause improper substitution of the preprocessed tokens, reproduce un-portable versions of the C-based programs, and represent them in the XML documents. Another limitation of our

\(^5\)The GNU Compiler Collection (GCC) is a compiler system that belongs to the GNU Project and that supports various programming languages [105].
tool is ignoring any embedded code from different languages (e.g., assembly). Moreover, the tool has a limitation with respect to the output XML documents. It represents a jump statement as an expression statement. However, the tool specifies the type of this expression statement as “Jump statement”.

In the following sections, we describe how to determine the transition groups of each statement and use them to describe the CDG.

3.6 Deriving the CDG

The CDG derivation involves a static analysis approach in which we identify the graph nodes and edges from the software architecture. In this section, we describe the CDG derivation methodology. We first define a number of architectural and transitional patterns. Based on the software architecture, we identify the execution paths among connections and construct the program CDG.

3.6.1 Architectural Patterns

An architectural pattern is a control flow structure associated with a class of language constructs, where the control flow depicts a specific behavior. We specify five architectural patterns and use them to describe the control flow structure of a software program. These patterns are selection, condition, loop, expression, and jump. A selection pattern represents a fan-out architecture fragment where the control may flow in different channels (e.g., switch case and if else statement). A condition pattern represents a control flow transition based on a logical condition (e.g., if statements with no else part). A loop pattern represents a cyclic architecture fragment where the control may iterate throughout a number of code statements (e.g., for, while, do while statements, and recursive procedure calls). An expression pattern represents a basic architecture fragment of an expression statement (e.g., assignment, unary expression, and procedure call). A jump pattern is like
an expression pattern except that its target architectural element is identified in a jump statement (e.g., \textit{goto}, \textit{continue}, \textit{break}, and \textit{return} statements).

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Nesting relation</th>
<th>Example Language constructs or rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$S \rightarrow L^n$</td>
<td>switch, if-else</td>
</tr>
<tr>
<td>Condition</td>
<td>$S \rightarrow L$</td>
<td>if</td>
</tr>
<tr>
<td>Loop</td>
<td>$S \rightarrow L$</td>
<td>for, while, do, recursive calls</td>
</tr>
<tr>
<td>Expression</td>
<td>$S \rightarrow \phi$</td>
<td>unary and assignment</td>
</tr>
<tr>
<td>Jump</td>
<td>$S \rightarrow \phi$</td>
<td>goto, continue, break, return</td>
</tr>
</tbody>
</table>

Table 3.2: Architectural patterns, nesting relations, and examples.

Table 3.2 lists the five architectural patterns and describes their control flow structures (nesting relations). The table also provides some example language constructs for each architectural pattern. The nesting of the patterns is a relationship from the statement domain ($S$) to the codomain of statement lists ($L$). The selection pattern associates one statement to one or more statement lists ($S \rightarrow L^n$). Both the condition and loop patterns associate one statement to exactly one list of statements (i.e., $S \rightarrow L$). The jump and the expression patterns do not associate a statement with any statement list ($S \rightarrow \phi$). Examples of these patterns are shown in the last column of the table.

3.6.2 Transitional Patterns

Program statements of each architectural pattern may pursue a number of control transitions during their execution according to the pattern structure. To outline the control flow structure among program statements based on these transitions, we specify a number of transition groups based on the aforementioned architectural patterns. A \textit{transition group} is a number of control transitions that can occur at a specific execution state according to an architectural pattern. To specify the possible transition groups of a program statement, we define a number of transitional patterns and describe the relationships among them. A \textit{transitional pattern} is an abstract class of transitions that represent a transition group. We introduce seven transitional patterns: pre-execution, inward, post-execution,
successive, iterative, deterministic, and block-end patterns. We use these patterns to determine the transition groups of each program statement according to these transitional patterns.

We use the running example of Figure 3.2 to provide some example transitions of these patterns. A pre-execution transition indicates a control transfer from a statement \( x \) in a block \( b \) through a connection to a statement \( y \) at the beginning of another (or same) block \( b' \) leaving behind the execution of the current statement until the connection returns (e.g., \( \text{tran}(15, 30) \)). An inward transition indicates a control transfer from a statement \( x \) to a statement \( y \) at the beginning of one of its associated statement lists (e.g., \( \text{tran}(15, 16) \)). A post-execution transition is similar to the pre-execution transition except that it occurs at the end of a statement execution (only for \( \text{do while} \) loop). A successive transition indicates a control transfer from a statement \( x \) in a code block \( b \) to a statement \( y \) that follows \( x \) in \( b \) (e.g., \( \text{tran}(15, 27) \)). An iterative transition indicates a control transfer from a statement \( x \) at the end of a code block \( b \) to a statement \( y \) at the beginning of \( b \) according to a loop statement (e.g., \( \text{tran}(23, 16) \)). A deterministic transition transfers the control from a statement \( x \) to another statement specified based on a \( \text{jump} \) instruction (e.g., \( \text{tran}(20, 23) \)). A block-end transition transfers the control from a statement \( x \) at the end of a code block \( b \) to the parent statement \( y \) that includes the block \( b \) (e.g., \( \text{tran}(18, 17) \)).

Each code statement can involve a number of transition groups corresponding to these transitional patterns according to its architectural pattern. Given a statement \( s_x \), we use these transitional patterns to classify all the possible transitions in this statement into transition groups. For example, the expression statement in Line 3 of the running example of Figure 3.2 includes 3 transitions. These transitions are pre-execute: \( \text{tran}(3, 10) \), pre-execute: \( \text{tran}(3, 13) \), and successive: \( \text{tran}(3, 4) \). Therefore, this statement includes 2
transition groups (pre-execute and successive), where these groups include 2 and 1 transition, respectively. The if statement in Line 23 includes 3 transitions. These transitions are inward: $\text{tran}(23, 24)$, iterative: $\text{tran}(23, 16)$, and block-end: $\text{tran}(23, 15)$. Therefore, this statement includes 3 transition groups (inward, iterative, and block-end), where each group includes only 1 transition.

### 3.6.3 Specifying the Transition Groups

We describe the control flow structure of a program statement based on these transition groups. We present a state diagram to describe the control flow among the specified transition groups. Then we list the state transfer conditions of this diagram and render the control flow structure of the running example of Figure 3.2.

**Control Flow Among the Transition Groups**

We describe the control flow structure of a program statement based on these transition groups. Given a program statement, we specify the control transfer through any transition of these groups based on the state diagram of Figure 3.8. In this diagram, the states are the transition groups and the edges are the state transfer conditions. The state diagram is important for deriving the CDG since it specifies the possible transitions based on a current state of a program statement. For example, the diagram answers the following question: if the last transition in a statement is inward, then which one is the next transition? It also answers the following questions: is it possible to execute an iterative transition after a pre-execute transition?, and for what type of architectural patterns this could happen? A state ($s_i$) in the diagram indicates the last executed transition type (i.e., pattern). Therefore, an edge in the diagram from $s_i$ to $s_j$ with condition $k$ can be read as follows, “if the last transition was $s_i$ (say inward), then, the next transition will be $s_j$ (say iterative), if the condition $k$ is true”. Figure 3.8 shows the 9 states: one for
each of the 7 transition groups in addition to a start state and an end state. The start state indicates that a statement has just received the execution. Please note from the figure that, a statement ends only when a deterministic, successive, or block-end transition occurs, since these states do not have any further transfer within this statement. In the diagram, edges are marked with the condition numbers listed in Table 3.3. From the state diagram, only two types of transitions can recursively occur: pre-execute and post-execute. This is due to the multiple number of calls that can exist in a statement and that should be executed sequentially one after another.
### Control Transfer Conditions Among the Transition Groups

Table 3.3 lists the state transfers of Figure 3.8. For each state transfer, the table describes the edge Id (column 1), “from” state (column 2), “to” state (column 3), architectural patterns allowing the state transfer (columns 4-8), and state transfer conditions (columns 9-13). The state transfer conditions (columns 9-13) are of type Boolean and they take the
values of ‘True’ and ‘False’. The 5 conditions are joined with logical ‘AND’ operation. Therefore, in order for a state transfer to occur, all the listed conditions should combine to ‘True’. Empty table cells in the condition (columns 9-13) indicate that the condition is not relevant or ineffective with respect to the state transfer occurrence. Empty cells in columns 4-8 indicate that the state transfer does not apply to the corresponding architectural pattern. For example, row 1 indicates that, after a statement starts execution it may transfer the control through an iterative transition (columns 1-3). This control flow structure can occur only with expression statement (columns 4-8). This control structure occurs only, if the statement is the last one in its block (column 11), where this block is a loop block (column 12) and the statement does not include any connections (column 13).

**Transition Groups and Control Transfers**

Table 3.4 displays the transition groups of the running example of Figure 3.2. The table rows are the code lines and the columns are the architectural patterns (column 2) and transition groups (columns 3-9). For example, if the statement in Line 15 is executed, then a pre-execution transition $\text{tran}(15, 29)$ represents the connection invocation of the procedure call $\text{peek()}$. After the return of the control to Line 15, an inward transition $\text{tran}(15, 16)$ transfers the control to the loop block. After executing the loop block, the control transfers from Line 15 to the successive statement in Line 27 through the transition $\text{tran}(15, 27)$.

**3.6.4 Building the CDG**

In combination with the software architecture, the transition groups can be used to build the control flow graph of a software program. Here, we describe our method to derive these transition groups and provide an algorithm to determine the CDG of a software program using these groups.
3.6. DERIVING THE CDG

In our methodology, we proceed in two steps of static analysis to determine the CDG of a program. In each step, we scan the whole syntax tree to capture some relevant information as shown in Table 3.5. The table displays the structural information obtained at each step about the software program. The table columns describe the field names, levels, steps, and description of these data elements. The level of a structural data element indicates the scope where the data element is associated in the program code (statement, CIB, or node). All data elements derived in step 1 are determined for each program statement, while the data elements in step 2 are derived at either the CIB level or at the CFG node level.

In step 1, we determine basic structural information about the software program and identify the transition groups of each statement. For each code statement, we generate

<table>
<thead>
<tr>
<th>Line</th>
<th>Arch.Pattern</th>
<th>Pre-execution</th>
<th>Inward</th>
<th>Post-execution</th>
<th>Successive</th>
<th>Iterative</th>
<th>Deterministic</th>
<th>Block-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Declaration</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Expression</td>
<td>10,13</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>10</td>
<td>Declaration</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Declaration</td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Expression</td>
<td>32</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Loop</td>
<td>29</td>
<td>16</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Expression</td>
<td>29</td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Loop</td>
<td></td>
<td>18</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Condition</td>
<td>19</td>
<td></td>
<td>18</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Expression</td>
<td>32</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>23</td>
<td>Condition</td>
<td>24</td>
<td></td>
<td>16</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Declaration</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15,16,34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Declaration</td>
<td></td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Expression</td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Expression</td>
<td>29</td>
<td></td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>

Table 3.4: Transition groups of the code example of Figure 3.2.
3.6. DERIVING THE CDG

Table 3.5: The information fields derived during the steps of rendering the CFGs.

<table>
<thead>
<tr>
<th>Data field</th>
<th>level</th>
<th>Step</th>
<th>Description or usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Id</td>
<td>Statement</td>
<td>1</td>
<td>A unique Id for each program statement</td>
</tr>
<tr>
<td>Block Id</td>
<td>Statement</td>
<td>1</td>
<td>The block Id where a statement belongs (if any)</td>
</tr>
<tr>
<td>Connector Id</td>
<td>Statement</td>
<td>1</td>
<td>An Id of the first connector (if any) in a statement</td>
</tr>
<tr>
<td># connectors</td>
<td>Statement</td>
<td>1</td>
<td>The number of connectors in a statement</td>
</tr>
<tr>
<td>Successors</td>
<td>Statement</td>
<td>1</td>
<td>Control flow successors of a statement</td>
</tr>
<tr>
<td>Start edges</td>
<td>CIB</td>
<td>2</td>
<td>The start edges of a CIB</td>
</tr>
<tr>
<td>Exit edges</td>
<td>CIB</td>
<td>2</td>
<td>The exit edges of a CIB</td>
</tr>
<tr>
<td>Subsequent edges</td>
<td>Node</td>
<td>2</td>
<td>The edges that originated at a CFG node</td>
</tr>
</tbody>
</table>

three identification numbers: statement Id, block Id, and connector Id (rows 1-3). These Ids are used to identify statements, statement blocks, and the first connector of a statement with respect to the execution priority sequence of a programming language. These Ids are generated as unique numbers within the program scope while tracking the whole syntax tree using a Breadth First Traversal (BFT) algorithm. They can also be used to specify the nodes of S-CFG, B-CFG, and CDG. Thus, these Ids are associated to each other according to the following rules.

Every statement is associated with a unique statement Id. A block Id is a unique Id for each branch-free code block and is associated with all statements that belong to it. A connector Id is a unique Id for each program connector and is associated with a statement only, if it includes any procedure call or event trigger. If the statement involves more than one connector, the Id of the first connector is selected and the number of connectors (row 4) is counted for this statement. These connectors include both the inter-component and the intra-component connectors in this statement. In this step, we also build a vector of subsequent statements (row 5) with respect to each code statement. This structural data element represents the transition groups of each statement according to its architectural pattern. The vector of subsequent statements is ordered according to the state transfer conditions as listed in Table 3.3 and the transition groups are specified as explained in Section 3.4.
3.6. DERIVING THE CDG

In step 2, we utilize the information determined in step 1 using an algorithm to identify the CDG nodes and connect the edges among them. The algorithm depends on

```c
/*-- main procedure --*/
void main()
{
    graph CDG = empty;
    CIB b = block of main()
    list L = b.statements;
    CreateNode(CDG.start, L.first(conn));
    CreateSuccessors (CDG.start);
    int p = 1;
    for each Component c {
        for each CIB:b in C) {
            if (not b.tracked)
                list L = b.statements;
                CreateNode(CDG.part[p], b.first(conn));
                CreateSuccessors (CDG.part[++p]);
        }
    }
    return CDG;
}

/*-- Create CDG node --*/
void CreateNode(Node n, Statement s)
{
    n.Tracked = True;
    Trans.Pattern TP=Start;
    for each(s.trans.group(conn)[1]: g) {
        for each(g.trans(conn)[j]: t) {
            if (Cond(s.AtrchP, g.transP))
                Node m = t.target;
                n.connect(m);
        }
    }

/*-- Create node successors --*/
void CreateSuccessors(Node n){
    for (each m in n.successors){
        if (not m.tracked){
            CreateNode(m, m.statements);
        }
        CreateSuccessors(m);
    }
}
```

Figure 3.9: A Depth First Search (DFS) algorithm for deriving a program CDG.
3.6. DERIVING THE CDG

the state diagram (Figure 3.8) and condition table (Table 3.3). The determined CDG in this step is stored in three data elements (rows 6-8) associated with each CDG node and CIB of the program. The CDG nodes are identified based on the connectors that may exist in the program statements. A node is created for each statement that includes one or more connectors (i.e., where its connector Id ≠ null). If a statement has more than one connector, then another set of CDG nodes are generated for the remaining connectors of this statement. The data elements in this step are the start edges, exit edges, and subsequent edges. The start and exit edges are determined for each CIB as illustrated in Figures 3.4(a) and 3.4(b), respectively. The subsequent edges are determined for CDG node from the transition groups of the corresponding statement. The derivation of these subsequent edges is accomplished using the CDG derivation algorithm presented in Figure 3.9. Thus for each CIB, these three data elements constitute the CDG of this CIB. In addition to the information of step 1 about each statement association with a block or a connector, we use the start, exit, and subsequent edges to derive the CDG for the whole program.

Figure 3.9 presents the CDG derivation algorithm from the transition groups described in Table 3.3. The algorithm uses Depth First Search (DFS) heuristic to derive a CDG. The DFS algorithm is implemented by traversing all CIBs one at a time. For each CIB, the algorithm creates its corresponding part of CDG and attaches that part to the previously traversed CIBs through the connectors in their code. The algorithm follows a recursive approach to scan all the statements of a CIB according to the code structure defined by the abstract syntax tree. In this algorithm, procedure main() (Lines 4-19) initiates and implements the DFS. The procedure CreateNode() (Lines 20-32) recursively creates a graph node and determines all of its subsequent transitions. This procedure loops on the transition groups of a statement $s$. Based on its transitional pattern and the statement architectural pattern, Line 26 ($\text{Cond}()$) checks the state transfer condition represented in
Table 3.3. However, this check does not consider any code semantics, therefore, it does not consider column 10 of the table which is meant to validate actual transitions. According to the state transfer condition, a successor of \(s\) may be created for each statement transition. The procedure \(CreateSuccessors()\) (Lines 33-41) recursively calls \(CreateNode()\) over the new graph nodes. After the algorithm traces all the software architecture, the CDG will completely represent the program code.

### 3.6.5 Deriving the Other CFG Structures

With the information obtained in steps 1 and 2 of the previous subsection, we are also able to determine the S-CFG, B-CFG, and the C-CFG. The S-CFG nodes are clearly identified as statements according to their statement Ids. B-CFG nodes are identified as the Ids of statements that are associated with a code block, by avoiding repetitions. If two consecutive statements have the same block Id, we only create one B-CFG node. C-CFG nodes represent program components by associating them with unique Ids. The edges of these CFG structures can be obtained by tracking the software architecture while considering edges among statements, code blocks, or connectors.

The information derived in step 2 corresponds initially to the S-CFG structure. By reducing the S-CFG edges among the statements that are not associated with code blocks, we are able to determine the edges among the B-CFG nodes. Similarly, by reducing the S-CFG edges among statements that are not associated with connectors, we are able to determine the edges among the CDG nodes. C-CFG is obtained by considering only the edges among the inter-component connections.

### 3.7 Experimental Evaluation

In the experimental evaluation of the CDG, we examine the effect of program size on the proposed CDG in comparison to the existing CFGs with respect to complexity and
3.7. EXPERIMENTAL EVALUATION

inaccuracy. We aim to derive the S-CFG, B-CFG, C-CFG, and CDG for different program segments with different sizes. For each derived graph, some complexity and inaccuracy factors will be determined. In the following subsections, we describe our experimental environment, process, and setup. Finally, we render our results.

3.7.1 Experimental Environment

For our experimental evaluation in this thesis, we choose PostgreSQL version 8.4.4 for its large size and availability of architectural design in a decent Web-based representation. PostgreSQL is an industrial open source object-relational database system that is used by hundreds of customers all over the world in several business fields including government, education, finance, retail, and telecom systems [95]. The PostgreSQL 8.4.4 database engine (the core server) includes 730 C code files. By considering each C-file in the PostgreSQL 8.4.4 database engine as a component, each component includes on average 23 CIBs. The source code of this application exceeds 900,000 lines of code and is written in C-language, assembly, and others. In our experiment, we run PostgreSQL on Ubuntu 11.04 operating system. The source code is analyzed and results are computed on Windows 7 operating system using Java 2 Platform, Enterprise Edition (J2EE). We derive the syntax tree of the PostgreSQL database and represent it in XML documents using our reverse architecting toolkit [104]. The PostgreSQL program structure and an example PostgreSQL program file (/backend/bin/pg_dump/pg_dump.c) are displayed in Figures A.3 and A.4. The following subsection describes the details of the experimental process.

3.7.2 Experimental Process and Setup

In our experimental process, we identify the software segments, components, and CIBs. For each program segment, we determine the CDG as well as the other CFGs. As indicated earlier, the C-CFG is determined by considering each program file as a component.
By inclusively grouping the software files in segments, we determine a number of overlapping program segments with different sizes. Therefore, the segment sizes in Lines Of Code (LOC) are incremental and proportional to the number of files in these segments. Thus, segments 1, 2, 3, \ldots, \( n \) include 1, 2, 3, \ldots, \( n \) components, respectively, and the number of segments is equal to the number of components. For example, segment 1 includes component 1 only, segment 2 includes component 1 and 2, and segment 3 includes component 1, 2, and 3, etc. While each segment is practically an incomplete sup-program and cannot run independently, the collection of segments will be beneficial to conduct the desired comparisons among the CFGs. The C-files that correspond to the components of a specific segment are considered as the only code contents of this segment. The procedure calls and event triggers among the components that belong to a segment are the only considered inter-component connectors in this segment with respect to its CFG representation. Other calls and event triggers to the components that do not belong to the current segment are not considered as connectors and only treated as local variables in each CIB. For each C-file, the CIBs are identified and associated with all program segments that include the corresponding component.

### 3.7.3 Comparing Graph Complexities

We present the results of our experimental evaluation. Figures 3.10(a) and 3.10(b) show the numbers of nodes and edges of each type of CFG (including CDG) and their relationships with the code size.

In these figures, the X-axis is the code size indicated by the number of components and the Y-axis is the number nodes and the number of edges in Figures 3.10(a) and 3.10(b), respectively. From Figures 3.10(a), it is clear that, the S-CFG always has the highest and the C-CFG always has the lowest number of nodes with respect to all the program sizes. Similarly, these two CFGs have the highest and the lowest numbers of edges,
3.7. EXPERIMENTAL EVALUATION

Figure 3.10: The numbers of nodes and edges of the CFG structures representing program segments with variables sizes.

respectively. The differences between the CFG complexities (indicated by the nodes and edges) increases with the code size. For example, the number of S-CFG nodes of the whole program (730 components) is more than double the number of B-CFG nodes, more than 7 times the number of CDG nodes, and more than 100 times the number of C-CFG nodes. The CDG always has relatively small number of nodes and edges in comparison to S-CFG and B-CFG and consequently, it is less complex than both of them. Relative to the C-CFG, the CDG is slightly more complex. However, this relatively tolerated complexity is a trade off with the gain of representing the inter-component and intra-component transitions with higher accuracy.

Figure 3.11: The occurrences of the edge types in the control flow graphs.
Figures 3.11 and 3.12 present a comparison among the different CFGs including the proposed CDG with respect to the types of edges in each graph. Figure 3.11 displays four pie charts, one for each CFG structure. A pie chart in this figure presents the ratios of edge type occurrences relative to the total number of edges in a CFG. In these charts, the vertical line, dark dotted, and light dotted sectors indicate the interface, loop, and transitional edges’ ratios, respectively, where a whole pie indicates the total number of edges in each CFG. In this chart, interface edge ratio includes the pre-execute and post-execute transitions. Loop edge ratio considers only the iterative transitions. Transitional edge ratio includes the inward, successive, deterministic, and block-end transitions. It is clear that, the S-CFG and B-CFG are congested with transitional edges among statements. Conversely, C-CFG does not present this type of edges by totally ignoring the internal component transitions. Unlike these CFGs, the CDG provides moderate balance between the transitional edges and the interface edges as it considers all system connections and all internal transitions among these connections.

Figure 3.12: The numbers of edges of each CFG structure and edge type averaged over the different program segments.

Figure 3.12 presents the numbers of edge types in these CFGs averaged over the 730 program segments. In this figure, the X-axis represents the CFGs. The Y-axis is the
average number of edges of each type, where these edge types are represented on the Z-axis. This chart allows comparing the average numbers of each edge type among the different CFGs. The chart shows that, the variances among these CFGs are high with respect to all types of edges. Of all the CFGs, C-CFG has the least number of edges, while S-CFG has the highest number of edges with respect to all types of edges. We also find that the CDG and the B-CFG considerably vary from each other with respect to all types of edges, where the CDG is less than half of the size of B-CFG.

3.7.4 Comparing Graph Inaccuracies

Figure 3.13 compares the CFGs with respect to their illegal branches. In Figure 3.13(a), the X-axis is the program size and the Y-axis is the number of illegal branches. The figure shows that the CDG always has fewer number of illegal branches than both B-CFG and C-CFG. Moreover, it shows that, the CDG mostly has less illegal branches than S-CFG, especially with respect to large program sizes.

![Figure 3.13](image)

Figure 3.13: The numbers of illegal control flow branches in the CFGs representing the program segments of variable sizes.

Figure 3.13(b) shows the illegal branch density (number of illegal branches per graph node) in each graph [89]. This illegal branch density describes the accuracy of a CFG according to its size. Unsurprisingly, S-CFG and CDG seem to be considerably more
accurate than C-CFG and B-CFG. S-CFG seems to have slightly less density of incorrect transitions relative to the CDG. However, this slight difference is a trade off with respect to the gain of representation simplicity in comparison with the S-CFG.

3.8 Summary

Three CFG structures are intensively used in software reliability analysis techniques: C-CFG, S-CFG, and B-CFG. The C-CFG is highly abstract and it only represents the inter-component control flow transitions. It is also vulnerable to illegal control flow sequences due to ignoring the dependencies among software connections. The S-CFG and B-CFG are extremely complex and are not suitable for large software systems. In this chapter, we propose a Connection Dependence Graph (CDG) for simple control flow representation that considers both inter-component and intra-component control transitions. A CDG is a connection-based control flow graph representation of possible connection invocation sequences that may occur during a program execution. Using our reverse engineering toolkit, we determine the software architecture from source code. We describe a number of architectural and transitional patterns and exploit them to capture the control transitions. Then, we categorize these transitions into a number of transition groups which are used to identify the execution paths among connections and utilize them to construct the CDG. We provide an experimental evaluation to examine the effect of program size on the CDG and the other CFGs by comparing them with respect to complexity and inaccuracy using the PostgreSQL open source database. Our results show that, the CDG always has relatively small number of nodes and edges in comparison to the S-CFG and the B-CFG. Regarding representation inaccuracy, the CDG always has fewer illegal branches than both B-CFG and C-CFG. Moreover, the results shows that, the CDG mostly has fewer illegal branches than S-CFG, especially with respect to large program sizes. In the following chapter, we utilize the CDG to provide a signature-based control flow monitor.
Chapter 4

A Signature-Based Control Flow Monitor

Control flow errors are major impairments to software system correctness. Existing signature-based error detection approaches rely on the Block-based CFG (B-CFG) to represent the system control flow structure. Validating the system execution transitions may be susceptible to false negatives due to the incomplete connection coverage of the B-CFG representation in these techniques. The connections that are invoked in a condition expression of a language control structure statement (construct) are not associated with any basic blocks since they exist in the start of a compound statement that may have several other branches. Likewise, if several connections exist in an expression statement, they are also not associated with any blocks since they cannot fit to a branch-free block. These ignored connections introduce CFG complexity through supplementary edges to represent their invocations as illustrated in Section 3.3. In addition, the existing techniques do not differentiate between connection invocations and execution transitions based on the language constructs. As a result, these techniques are hard to use for backtracking program execution trace and validating its successful termination. Moreover, the branch-free block partitioning in these techniques decreases the precision of error localization inside these blocks.
4.1. Overview

In this chapter, we utilize the Connection Dependence Graph (CDG) representation to construct a connection-based signature approach for control flow error detection. In this approach, we utilize the connection-based control flow signature structure in which, we partition the program components into Connection Implementation Blocks (CIBs) (Section 4.2). We provide the control flow monitor structure and explain the control flow error checking algorithm based on these CDGs (Section 4.3). An experimental evaluation of the proposed approach is provided using PostgreSQL open-source database [95].

4.1 Overview

Figure 4.1 presents an overview of the proposed technique in two steps: preparation and monitoring. In the preparation step, we instrument the software system in order to enable the generation of runtime signatures during its execution. A connection-based control flow signature is a runtime state describing the currently invoked connection during a system runtime. A program is partitioned into CIBs. Each CIB is represented using a CDG. These CDGs are used to compute the runtime signatures. Signatures are injected in the program code to allow exporting the actual runtime transitions in a form of runtime signatures. In the monitoring step, we capture these runtime signatures and use them to verify the system control flow transitions. The monitor validates these runtime signatures.
by using the valid control flow transitions specified by the program CDGs and finally, an error report is generated.

4.2 Connection-Based Signature

In a connection-based signature structure, a unique signature is given to each connection of a software program. In this section, we partition the program into CIBs. For each CIB, a CDG is obtained from its code block to represent its control structure (Subsection 4.2.1). We finally describe our signature form and its injection method (Subsection 4.2.2).

4.2.1 Program Partitioning

During a program runtime, the currently executing component and the current Invocation Code Blocks (ICBs) are important ingredients to identify a connection in our signature structure. To map each connection to its component and ICB, we first provide a unique Id for each program component. We then partition each component according to its CIBs. We use these CIBs as ICBs (see Subsection 2.1.1) to describe the runtime signatures with respect to the connection invocations. Some other code segments of the program may exist outside these CIBs. However, these code segments are usually preprocessing, prototyping, and variable declarations which do not involve any connection invocations. Consequently, they are not assigned any signatures in this technique. We use these CIBs as program partitions and represent the code of each CIB using a separate CDG as explained in Chapter 3.

To enable the differentiation between connection invocations and all other execution transitions, we reduce the CDGs of the program CIBs by omitting the connection invocation and return representations. Therefore, the edges of these CDGs represent only the control transitions among connector nodes. The nodes of the reduced CDG implicitly represent the connection invocations and the edges represent all other transitions. This
differentiation is important to track inter-component interactions and to create a call (or execution) stack.

Based on this partitioning and reduced CDG representation, we specify two types of control flow transitions relative to any CDG node: connection invoker and connection passer. Relative to any CDG node, a connection invoker indicates the target CIB. A connection passer indicates the subsequent node transition in the current ICB. However, a connection invoker always precedes all subsequent transitions of a CDG node. If the target CIB of a connection returns, then, a transition through any edge of this node (connection passer) can take place. Using the CDG, a system state is described by the invoked connector and a state transition indicates a transfer of the execution among these connectors.

![CDG Representation](image)

Figure 4.2: The CDG representation of the program partitions representing the code example of Figure 3.2.

By implementing this partitioning on the example code of Figure 3.2, 5 CIBs are obtained as shown in Table 3.1. These CIBs are $cib(1, \text{main})$, $cib(2, \text{readAdjMat})$, $cib(2, \text{dfs})$, $cib(2, \text{peek})$, and $cib(2, \text{attachNode})$. Each of these CIBs is represented by a separate CDG. Figure 4.2 presents these CDGs and illustrates the differentiation between the connection invokers and passers. The CDG edges represent only the connection passers while the
connection invokers are omitted and only considered implicitly. For example, two transitions are shown in dashed lines as sample omitted connection invokers. These connection invokers are $\text{tran}(3b, S_3)$ and $\text{tran}(E_3, 3b)$, where $S_3$ and $E_3$ are the start and end nodes of $\text{cib}(2, \text{dfs})$. If node $3b$ is executed, then, first, the connection invoker $\text{tran}(3b, S_3)$ is executed. Once the connection returns through $\text{tran}(E_3, 3b)$, the control transfers to node $E_1$ through $\text{tran}(3b, E_1)$.

### 4.2.2 Signature Form and Injection Method

A runtime signature is a unique string that describes system execution state with respect to its current control flow transition. A signature string identifies the current executing component, CIB, and connector. Consequently, a signature is formed as “Component Id, CIB Id, Connector Id”. For example, the signature “163, 9, 5” indicates that component 163 is executing the 9-th CIB and invoking the 5-th connector in this code block. If several connections exist in one code statement, a compound signature is used to describe the system state at the time of executing this code statement. An example compound signature is “163, 9, 5; 6; 7”. This signature indicates that the currently invoked connector exists in a statement that includes other connectors. The invocation order of these connectors is: “5, 6, 7”. Each signature is injected in the program code according to the invoked connector statement location. We inject a line of code before each statement that includes one or more connectors. The injected code line writes the included signature to an output file stream.

Based on this signature structure, we illustrate how to assign signatures for the code example of Figure 3.2. As shown in Table 3.1, the example program includes 2 components, 5 CIBs, 9 statement lists, and 7 connectors. These 7 connectors exist in Lines 3, 14, 15, 16, 19, and 35. A signature is injected before each of these files as shown in Table 4.1. The table displays the code line that includes connections in the first row and
4.3 Control Flow Monitor Structure

The error detection technique uses a CDG to represent a program architecture. A runtime monitor uses this CDG to validate the control flow transitions among system connectors. In this section, we describe the proposed monitor structure as shown in Figure 4.3. The instrumented software systems generate runtime signatures representing the actual control flow transitions. The monitor receives these signatures and validates them using the stack manager and the error detector. The stack manager maintains a control stack for the connection invocations during system runtime. The error detector validates runtime signatures by comparing them to the CDG by using the control stack. If errors are found, the monitor generates an error report. The use of control stack allows detecting not only illegal branches but also unreturned connections in case of an improper system termination. In the following subsections, we explain the details of the control stack and the error detection algorithm. Then, we describe the detected error state parameters.
4.3. CONTROL FLOW MONITOR STRUCTURE

Figure 4.3: The control flow monitor structure.

4.3.1 Control Stack

The control stack tracks the connection invocations of a system during its runtime to allow validating connection return transitions. In case of an improper system termination, this stack helps detect incomplete execution traces and unreturned connections. The control stack includes a list of signatures describing previously invoked but not yet returned connectors. Each stack element indicates a connection invocation that took place inside the CIB of the preceding stack element. A connector in the stack element at the i-th position can return only if all the connectors in the higher stack positions return beforehand.

In addition to the known three stack operations push(), pop(), and peek(), we perform two more management operations to reduce repeated element lists: attach() and detach(). These operations help decrease the stack size if it includes any repeated lists of elements. These repetitions are usually caused by loop structures and recursive procedure calls. attach() and detach() operations merge repeated stack portions and split merged portions, respectively using some reduction information stored in an array repository. attach() is used for pushing new stack elements and detecting repeated stack parts. detach() is used
4.3. CONTROL FLOW MONITOR STRUCTURE

for removing a stack element that exists in a merged stack portion.

The reduction array \( a^R \) is structured as \( a^R = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_N, y_N)\} \). The array cells (from left to right) partition the reduced stack elements successively (from top to bottom) into a number of sub-lists \( N \). A cell number \( n \) is mapped to a list of elements in position \( n \). A cell in this array is an ordered pair \((x_n, y_n)\), where \( x_n \) is the number of stack elements in the \( n \)-th list and \( y_n \) is the number of repetition of this list. For example, consider a stack \( S = \{s_1, s_1, s_2, s_1, s_1, s_3, s_1, s_3\} \) that includes some repeated portions where the leftmost element \( s_1 \) is the top of the stack. By using attach() operation, the stack can be reduced to \( S^R = \{s_1, s_2, s_1, s_3\} \). The reduction information of this stack is maintained in an array \( a^R \), where \( a^R = \{(1,1), (1,0), (2,1)\} \). This reduction array indicates (from left to right) that the topmost element \( (s_1) \) of the reduced stack \( S^R \) exists once. Then the next element \( (s_2) \) is not repeated. Finally, the last two elements \( (s_1, s_3) \) are repeated once. This interpretation of the reduced stack \( S^R \) allows restoring the original stack \( S \) using the detach() operation.

4.3.2 Error State Parameters

Two types of control flow anomalies are detected in our work: illegal branches and unreturned connections. As defined in Subsection 2.1.3, an illegal branch indicates a control transfer that does not comply with the software architecture represented using the CDGs. Unreturned transitions indicate improper program termination. A program terminates appropriately, if the control stack is empty or if it exits using an “end” operation (e.g., “system.exit()” library function in C-language’). In case of an improper exit of a program, the remaining stack elements are considered as control flow errors and classified as unreturned connections.

The other error parameters determined in this work include error characteristics: external, repeated, and stacked. An external transition is a control transfer to a CIB in
a component that is different from the current component. A repeated transition is a control transition that has been classified as valid in an earlier system state with different stack elements. A stacked transition indicates that the target runtime signature of the incorrect transition already exists in an element that is not on the top of the stack. Error location is another parameter that is determined using our technique. This parameter is directly derived from the runtime signature. Using the error location, one can determine the most erroneous or most intact system components.

The determined error parameters provide important comprehension about control flow error states. For example, unreturned connections may indicate silent failures. Illegal branches can be used to predict other failure types (e.g., value or performance failures) depending on the values of the other parameters: external, repeated, and stacked. The determined error parameters provide important comprehension about a system control flow error state. These error parameters can also be used to determine the components that are highly correlated with performance delay. Figure A.10 displays a sample of the monitor output of observing a single system run. In the next chapter, we will utilize these error state parameters to predict failure occurrences and forecast their modes.

4.3.3 Control Flow Error Detector

The control flow error detector is an algorithm that runs each time a runtime transition is observed by the monitor. The algorithm relates the current system state to the program CDG and then compares the runtime signature to the possible valid transitions based on this CDG. The detector interacts with the stack manager by adding or removing elements. In case a proper termination using a `system.exit()` operation is observed, the algorithm triggers emptying the stack to avoid reporting its remaining elements as unreturned connections.
4.3. CONTROL FLOW MONITOR STRUCTURE

Figure 4.4: The control flow error detector.

The algorithm in Figure 4.4 presents our control flow checking method. The algorithm checks the system control flow for illegal branches (Lines 3-20) while the system is running and for unreturned connections (Lines 21-25) after the system exits. Line 3 checks the system running state. If the system is running, Lines 4-6 determine if the current transition is system.exit() and consequently, clear the stack if it is true. Lines 7-18 validate the actual transition $t$ based on the current system state $s$. First, Line 7 determines the CIB currently executed from the system state $s$ using the component and CIB IDs of the state $s$. Line 8 determines the current node in the CDG of the currently executed CIB using the connector Id of the state $s$. Lines 9-18 validate the transition $t$ against the invoker edge of the current CDG node $n_i$. If the transition is validated, then $s$ is pushed into the stack for future control return checking. If it is not validated against the connection
invoker, Lines 12-17 validate the transition $t$ using the control stack. Line 12 peeks the top node of the control stack. Line 11 validates the transition $t$ against the top stack node. If it is validated using these connection passers, the stack top node is popped out. Otherwise, an error report is generated. In Line 9, the transition $t$ becomes the current state $s$. Finally, Lines 21-25 generate an error report for the unreturned connections that are remaining in the control stack.

4.4 Experimental Evaluation

We evaluate the error detection technique using the PostgreSQL 8.4.4 database software system. We aim to examine the efficacy of our approach to detect control flow errors in a number of software system versions which are altered randomly to produce different numbers of control flow errors during runtime. For this purpose, 10 different program versions were generated by randomly altering some code statements and altering the program control flow structure accordingly, without introducing compilation errors. Each version is operated and monitored using 10 different experimental data sets. The experimental environment is explained in Subsection 3.7.1. In the following subsections, we describe the experimental process and setup. Then, the latter subsections present the results.

4.4.1 Experimental Process and Setup

Prior to generating signatures and injecting them among other logging code, we use our reverse architecting toolkit [104] to compute the CDGs of the program CIBs. A unique signature is given to each connector based on the component Id and the block Id that includes it. We inject a line of code to export the current execution state as a runtime signature. A similar line of code is injected prior to each connector in the program (see the example in Figure A.5). The injected code lines simply call a procedure implemented in a C-file that have been added and associated with a GNU Make file of the PostgreSQL
project (see the example in Figure A.6). This procedure manages the logging information in separate transition repository files and assures that the number of logs in each file does not exceed 100,000 transitions. It also arranges the transition repository files according to the process Ids generated by the forking (concurrent execution) mechanism of the PostgreSQL database (see the example in Figure A.9). The database system is executed and the control flow transitions are validated according to the Algorithm in Figure 4.4. Depending on the current stack of each transition, control flow error parameters are determined for each error. The detected errors are exported in log records and stored in a file repository. In the following paragraphs, we describe our methods for generating different software versions and for automatic software operation.

**Software Version Generation**

Ten versions of PostgreSQL database with different numbers of altered code statements are created using our toolkit [104]. Statements were altered in random components and random code locations. Two methods were applied to alter code statements randomly: repetition and removal. The repetition method duplicates a code statement and the removal method deletes a statement. The altered statements are selected randomly with some constraints to avoid compilation errors and to control the impact of the altered statements on the control flow. The chosen constraints assure that only expression statements are altered. They also avoid altering a statement, if it is the only one in its block and avoid repeating jump statements. The numbers of altered statements in these versions are chosen as 0, 3, 6, ..., 27 for versions 0, 1, 2, ..., 9, respectively. This incremental sequence of altered statement numbers is chosen to check their variable impact on the control flow errors of these versions. Note that, version 0 has 0 altered code statements, therefore, this version is considered to be the original version and is used to validate the outputs of the other versions with respect to the same inputs.
Automatic Software Operation and Experimental Data Sets

An automatic operation toolkit was developed to allow configuring, compiling, and building the executable files of the different software versions (see Figure A.7). The toolkit also cleans previous runs' information and prepares for receiving new transition repository files. After performing these setup procedures, the toolkit operates each version individually against all the experimental data sets and arranges the transition repository files in a folder structure. The tool includes a number of shell programs operating on Red Hat and Ubuntu Linux systems.

Experimental data sets are chosen to allow maximum coverage of the database engine operations and to allow automatic execution of the different versions against these data sets. Ten different experimental data sets are created to cover the server management, database management, client application management, data management, and tool management of the database engines. These data sets are created in the form of SQL files and executed as parameters to the database engine in the command lines of the shell programs (see Figure A.8). By including different lists of database commands in these SQL files, the provided experiments vary with respect to their performance loads. Table A.1 displays the actual execution times of system runs against these experimental data sets. Most of the data sets operate the system from 1 to 25 seconds. The experimental data set \( E_9 \) executes the system for significantly longer time. The different performance loads allow comparing system execution times. The large experimental data set \( E_9 \) aims to increase the executed code coverage during system runs.

4.4.2 Resulting Errors in Versions and Runs

Figure 4.5 shows the control flow errors detected in the different versions against the experimental data sets. Figures 4.5(a)-4.5(j) display 10 runs (0 to 9), where run 0, 1, 2, ..., and 9 exploit the data sets 0, 1, 2, ..., and 9, respectively. In these figures, the X-axis
4.4. EXPERIMENTAL EVALUATION

represents the versions 0 to 9 and the Y-axis represents the total number of control flow errors. The result of each run is displayed separately to ensure the visibility of the graphs since each run has a different vertical scale range. Figure 4.5(k) displays the results from all the runs to allow comparing them.
In general, these graphs show the increase in the control flow errors with the increase in the number of altered statements from version 0 to version 9. Version 0 is not mutated and it represents the original version. Versions 1-5 included approximately 0, 131, 144, 122, and 163 control flow errors, respectively on average among all the runs. Versions 6-10 included relatively higher average of control flow errors in most versions. These versions approximately included 1072, 228, 94, 2512, and 100 control flow errors, respectively on average among all the runs. In these versions, we notice that versions 6 and 9 have relatively higher control flow errors than the other versions and versions 8 and 10 have lower control flow errors than the others. The reason behind the increase of control flow errors in versions 6 and 9 is the occurrence of errors in the first run (run 0). This run mainly initiates the database server (Postgres). The errors in this particular operation are always carried out across all the remaining experimental data sets. The errors in the server startup mean that there are some server processes that failed to initialize. Consequently, these initialization failures will generate more failures in any further use of these processes. As a result, we notice that versions 6 and 9 have the highest number of errors. Conversely, versions 8 and 10 have the lowest number of control flow errors. This is due to the randomness of the statement altering method. It is noticeable that the altered statements in these versions are not affecting the control flow as much as other versions. This may be due to altering statements in unused or slightly used components with respect to the selected experimental data sets. It may also be due to the importance of the selected statements or to the statement altering method within the selected components.

4.4.3 Resulting Error Parameters

Figure 4.6 presents the number of control flow errors according to the error types, characteristics, and highest occurrences in system components. The error types (illegal, unreturned) and characteristics (external, repeated, and stacked) are shown in Figure 4.6(a)
4.4. EXPERIMENTAL EVALUATION

Figure 4.6: The control flow error types, characteristics, and highest occurrences in system components.

and 4.6(b), respectively. The X-axis represents the versions 0 to 9 and the Y-axis represents the total number of control flow errors. It is clear that, most of the detected control flow errors are illegal branches for connections to other system components (external transitions). The number of unreturned connections is relatively low and most of the detected errors are related to the transitions that target components which do not exist in the current stack. Most erroneous transitions are not repeated, i.e., they attempt to transfer the system to states that have not occurred previously in the current runtime.

Figure 4.6(c) presents the control flow errors according to the error location. The top
10 components with the highest control flow error occurrences are shown in the figure based on their component Ids. Components 165, 280, 469, 276, 63, 176, 519, 465, 267, and 252 correspond to dllist.c, pgstat.c, dynahash.c, autovacuum.c, xlog.c, pqsignal.c, aset.c,elog.c, sysv_sema.c, and scan.c, respectively. However, this high number of errors only indicate that the altered statements in the different versions are mainly distributed over these components. It also indicates that these components are mostly used by the experimental data sets during the system operation. The figure shows that our technique is able to locate the responsible component of each control flow error. Our technique can further narrow down the location to the code statement at which an illegal branch or unreturned connection has taken place.

4.5 Summary

Signature-based control flow monitors do not require any additional hardware or software redundancy. However, existing signature-based control flow monitoring approaches rely on the B-CFG to represent the system control flow structure. Therefore, in these techniques, validating the control flow transitions may not be feasible for large software systems and may be susceptible to false negatives due to the inclusion of some illegal control flow branches in the B-CFG. In comparison to the existing techniques, the use of the CDG as underlying control flow representation can decrease the false negatives in signature-based error detection. In this chapter, we propose a runtime monitoring approach for control flow error detection using connection-based signatures. This signature structure covers more program connections and associates each one to a unique signature. We describe the connection-based signature structure, explain how to partition a program into CIBs, and associate a CDG to each partition. We provide our control flow monitor structure and checking algorithm based on these CDGs of program partitions. An
experimental evaluation of the proposed approach is provided using PostgreSQL open-source database. The results show that our technique is capable of detecting control flow errors in 10 different versions with variable incremental numbers of randomly altered program statements with respect to the software control flow structures. The detected control flow errors in these versions are found to be mostly increasing with the number of altered program statements. The control flow monitor is able to detect the improper system termination by distinguishing the unreturned connection invocations. Regarding the error state parameters, most of the detected control flow errors are due to the illegal branches of external transitions and they are also not repeated. In the following chapter, we utilize the detected error state parameters to provide a prediction approach for failure occurrences and modes during system runtime.
Chapter 5

Runtime Prediction of Failures from Error Logs

Predicting failures in an early stage of their manifestation is important to assure system resilience and to minimize the costs of these failures. In Chapter 3, we provide a simple representation of software control flow (CDG) with lower number of illegal control flow sequences. In Chapter 4, we utilize this CDG to construct a signature-based control flow monitor that aims to decrease the false negative error detections. The monitor detects errors and characterizes a number of erroneous state parameters. In this chapter, we utilize the detected errors and state parameters to predict failure occurrences and modes during system runtime.

Our methodology utilizes system error log records to craft error spread-signatures that summarize error occurrences throughout system runtime (Section 5.2). An error spread-signature is a runtime state describing the error distribution among erroneous state parameters and error increase during a system run (Subsection 5.2.1). Using system error log history, we determine a predictive function (estimator) for failure mode eventuality based on these signatures (Subsection 5.2.4). A failure mode is a manner of failure manifestation [6]. A failure mode eventuality indicates the probability of occurrence of that failure mode. We show how to resolve the estimator and determine the predicted failure occurrence and mode in correspondence to each log record (Subsection 5.2.5). An experimental evaluation using PostgreSQL open-source database is provided (Section 5.3).
5.1 Overview

Figure 5.1 outlines our failure prediction method in two phases: preparation and forecasting. In the preparation phase we use the system error log history and failure history to determine a number of failure mode estimators using a regression analysis method. In this analysis, we craft an error spread signature and calculate a failure mode eventuality measure for each log record. The crafted signatures and eventuality measures are utilized by using an Ordinary Least Squares (OLS) regression procedure to determine the failure mode estimators as coefficient matrices of the error spread signature variables. We determine an estimator for each failure mode that has occurred during system history. In the forecasting phase, we utilize these estimators to predict the failure mode eventualities from the error log during system runtime. Depending on the eventuality values of each failure mode, we use a threshold parameter to resolve the failure prediction results. In correspondence to each log record, the failure prediction result (an estimated failure occurrence and mode) is exported to a failure report.

5.2 Failure Occurrence and Mode Prediction

Failures may occur during a system runtime and their modes differ with respect to the manifestation manners. The propagation of errors and the change in error state parameters along a system runtime are obvious characteristics of these manifestation manners.
In this section, we present our approach for software failure occurrence and mode prediction. We describe and show how to derive the error spread-signature and the failure mode eventualty from a log record. Using regression analysis, we derive a predictive function (estimator) for failure mode eventualties based on these error spread-signatures and show how to resolve the failure occurrence and mode results. Finally, we discuss the performance overhead of our prediction technique and summarize the regression and prediction processes.

5.2.1 Runtime Error Spread-Signature

During the execution of a software system, numerous information elements about system events, rebellious behavior, performance counters, and error parameters are strictly monitored and audit trails are recorded in system logs. These system logs contain a wealth of information about system execution and erroneous states. Using these system logs, potential failures can be predicted without any knowledge about software design and any need for software instrumentation. Similar to [18, 21, 106, 107], we assume the adequacy of system log data to describe system error occurrences and erroneous state parameters.

Depending on the scope of system log information, a log trail can involve several data elements characterizing runtime intervals. Examples of system log data include CPU and memory utilization, disk read/write operations, current executing component(s) or subsystem(s), control stack, error counter, error messages, error types, and error state parameters. Such log data is of vital interest to failure prediction, as it reflects the system’s behavior and execution conditions. In practice, reliability analysts use this system log data to identify the counter behavior by comparing them to pre-specified thresholds of a specific failure domain. For example, if the memory utilization exceeds a number of Giga Bytes (GBs), a system is susceptible to performance failures. The change of these error log details during a system runtime characterizes the different manifestation manners of
failure modes. In this technique, we aim to capture a runtime error spread-signature based on the error distribution among the error parameters and error increase during the system runtime.

We express the error distribution among the error parameters in the runtime signatures using error log variables. An error log variable expresses a counter of a specific error type, message, or parameter in the system log. A signature can be constructed based on one or more of these variables depending on their availability in the log trail. We express the increase in error counters along the system runtime by summarizing the observations of a log variable in all preceding log records using a statistical summary function\(^1\). This summary function is utilized to craft the runtime signatures as a vector of elements in the following form.

\[
s_n = (\upsilon^1_n, \upsilon^2_n, \ldots, \upsilon^J_n)
\]

where \(s_n\) is the runtime signature of log record number \(n\). Each signature variable \(\upsilon^j_n\) represents a statistical summary of an error log variable \(g^j_n\) throughout the previous system error log trail (Records 1 to \(n\)). To incorporate the change in error counters among the log records, we calculate the statistical summary of these counters as the cumulative sums in the sequence of log records. Therefore,

\[
s_n = \left( \sum_{k=1}^{n} g^1_k, \sum_{k=1}^{n} g^2_k, \ldots, \sum_{k=1}^{n} g^J_k \right)
\]  \hspace{1cm} (5.1)

where \(\upsilon^j_n\) represents the summation of the log variable \(g^j_n\) observations from the start of the system run until record number \(n\). The error log data may need some preparation to avoid incomplete log records, conversion, normalization, and/or interpretation.

\(^1\)Several statistical summary functions can be used to describe or summarize a number of observations. Examples of these functions are summation, mean, deviation, and variance.
5.2. FAILURE OCCURRENCE AND MODE PREDICTION

5.2.2 Generating Runtime Signatures from Error State Parameters

We use the results of our control flow monitor (Chapter 4) to generate system runtime error spread-signatures. The monitor characterizes illegal control flow branches with respect to their target procedures (external or local), previous occurrence (repeated or not), and the existence of the control transition in the control stack (stacked or not) (see Subsection 4.3.2). An error state pattern is a vector of all possible combinations of error state parameter values. Errors are counted separately for each error state pattern and reported as log counters in the system log trails. Figure A.10 displays a sample of the monitor output after observing a single system run.

<table>
<thead>
<tr>
<th></th>
<th>(\mu_1)</th>
<th>(\mu_2)</th>
<th>(\mu_3)</th>
<th>(\mu_4)</th>
<th>(\mu_5)</th>
<th>(\mu_6)</th>
<th>(\mu_7)</th>
<th>(\mu_8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>External transition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Repeated transition</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stacked transition</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1: The possible error patterns determined using the error state parameters.

We use the accumulated summation of these error counts along an error log trail as signature variables according to Equation 5.1. By using these error characteristics, Table 5.1 lists the error patterns determined using these error state parameters. The table lists the error state parameters in column 1 and the patterns in columns 2 to 9. Each row represents the possible truth combinations of the values of the corresponding error parameter. Thus, using these error state parameters, there are possibly \(2^3\) state patterns that could characterize an error state: \(\mu_1, \mu_2, \cdots, \mu_8\). Using these error state patterns, we can use the error count of each pattern as a log counter variable and create an 8 element signature vector for each error log record. The error counter of a specific pattern describes the number of error occurrences since the time of the previous error log record until the time of the current log record.

\[
g_k^i = \text{count}(\mu_i, k) \quad (5.2)
\]
where $g_i$ is the i-th signature variable, $\mu_i$ is i-th error state pattern, and $count(\mu_i, k)$ is a function that counts the number of errors with respect to a pattern $\mu_i$ in a log record $k$. This signature variable can be ready to calculate runtime signatures using Equation 5.1 as shown in Table 5.2. In the following subsections, we describe the failure mode eventuality and how to predict it from the runtime signatures.

### 5.2.3 Failure Mode Eventuality

The purpose of our regression analysis is to predict a failure mode eventuality in response to each runtime signature. Failure mode eventuality indicates the probability of a failure mode during the system execution. We represent the failure mode eventuality of a system at a specific time as a normalized error count in the interval $[0, 1]$. A zero eventuality indicates that a failure mode $m$ is not currently expected. An eventuality of 1 indicates that a failure of mode $m$ has already occurred at system runtime. To obtain this measure for each log record $n$ during system runtime, we use the cumulative error sum until record $n$ as follows.

$$h_m^n = \frac{error(n)}{error(N)} \times bin(m)$$

(5.3)

where $h_m^n$ is the eventuality of failure mode $m$. $error(n)$ and $error(N)$ are the cumulative error sums since the start of the system run until log record number $n$ and $N$, respectively. $N$ is the total number of log records of a system run. $bin(m)$ is a binary value representing the occurrence of a failure mode $m$. Therefore, the values of $h_m^n$ can be described as follows.

$$h_m^n = \begin{cases} 
0 & \text{if } n = 0 \land m = false. \\
\in [0,1] & \text{if } n \in ]0,N[ \land m = true. \\
1 & \text{if } n = N \land m = true.
\end{cases}$$

(5.4)

We aim to monitor the system execution with respect to a number of failure modes
These failure modes can be selected according to the preferences of the reliability analyst from the system failure history.

Figure 5.2 illustrates the failure mode eventuality values that are predicted during a system execution. For each known failure mode, an estimator allows predicting its eventuality from a current log record. Therefore, the latest predicted eventuality of each mode (indicated as triangular pointers in the figure) will be monitored throughout a system runtime. A self-configuration component can perform a corrective action based on these eventuality pointers prior to a failure occurrence.

In the following subsection, we aim to determine an estimator $\alpha^m$ which can be used to estimate the eventuality of a failure mode from system error spread-signatures, i.e., $\alpha^m(s_n) = h^m_n$.

Table 5.2 displays the log record signatures of Experiment 7 (Run 6) shown in Figure A.10. The table lists 21 log records and their corresponding signatures. The table also displays the failure mode eventuality of each log record. In this example, the eventualities of two failure modes ($m_1$ and $m_2$) are calculated by assuming that the second failure mode has occurred at system runtime.

### 5.2.4 Determining Failure Mode Estimators

Regression analysis involves statistical techniques for modeling the relationship between one or more independent (observed) variables and a dependent (predictor) variable. Here,
5.2. FAILURE OCCURRENCE AND MODE PREDICTION

Table 5.2: The error counters, error spread-signatures, and failure mode eventuality measures of the example runtime log shown in Figure A.10.

<table>
<thead>
<tr>
<th>Rec.</th>
<th>Error counter</th>
<th>Signature</th>
<th>Eventuality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu_1 \mu_2 \mu_3 \mu_4 \mu_5 \mu_6 \mu_7 \mu_8$</td>
<td>$v_{n_1}^1, v_{n_2}^2, v_{n_3}^3, v_{n_4}^4, v_{n_5}^5, v_{n_6}^6, v_{n_7}^7, v_{n_8}^8$</td>
<td>$m_1=F, m_2=T$</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 0 0 0 0</td>
<td>1 1 0 0 0 0 0</td>
<td>0 0.0175</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 1 0 0 0 0 0</td>
<td>0 0.0175</td>
</tr>
<tr>
<td>4</td>
<td>1 1 0 0 0 0 0 0</td>
<td>2 2 0 0 0 0 0</td>
<td>0 0.0351</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 0 0 0 0 0</td>
<td>2 2 0 0 0 0 0</td>
<td>0 0.0351</td>
</tr>
<tr>
<td>6</td>
<td>5 5 0 1 0 0 0 0</td>
<td>7 7 0 1 0 0 0</td>
<td>0 0.1316</td>
</tr>
<tr>
<td>7</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 7 0 1 0 0 0</td>
<td>0 0.2193</td>
</tr>
<tr>
<td>8</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 0 1 0 1 0 9 1</td>
<td>0 0.2456</td>
</tr>
<tr>
<td>9</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 3 1 0 1 9 3</td>
<td>0 0.3246</td>
</tr>
<tr>
<td>10</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 3 1 0 1 9 3</td>
<td>0 0.3772</td>
</tr>
<tr>
<td>11</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 7 1 2 2 9 3</td>
<td>0 0.3860</td>
</tr>
<tr>
<td>12</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 7 1 2 2 9 3</td>
<td>0 0.3860</td>
</tr>
<tr>
<td>13</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 7 1 2 2 9 3</td>
<td>0 0.3860</td>
</tr>
<tr>
<td>14</td>
<td>0 0 0 0 0 0 0 0</td>
<td>7 13 7 1 2 2 9 3</td>
<td>0 0.3860</td>
</tr>
<tr>
<td>15</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 8 1 3 2 9 3</td>
<td>0 0.4737</td>
</tr>
<tr>
<td>16</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 8 1 3 2 9 3</td>
<td>0 0.5614</td>
</tr>
<tr>
<td>17</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 10 5 10 13 3</td>
<td>0 0.6316</td>
</tr>
<tr>
<td>18</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 10 5 10 13 3</td>
<td>0 0.7105</td>
</tr>
<tr>
<td>19</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 10 13 13 13 5</td>
<td>0 0.8246</td>
</tr>
<tr>
<td>20</td>
<td>0 0 0 0 0 0 0 0</td>
<td>10 18 10 13 13 13 5</td>
<td>0 1.0</td>
</tr>
</tbody>
</table>

we utilize system log history to model the relationships between system runtime error log and failure mode eventualities and use these relationships as predictive functions for future runtime failure modes.

Several regression procedures have been used intensively in the current literature for predicting and forecasting. Examples are the Ordinary Least Squares (OLS), the Total Least Squares (TLS), the Least Absolute Deviation (LAD), and the Maximum Likelihood Estimation (MLE). In this work, we use the OLS procedure for its computational simplicity and widespread usage in econometrics, control theory, signal processing, among other areas of applications. The OLS method is a generalized multiple regression model that aims to minimize the sum of squared vertical distances between the observed and the predicted variables by an estimator [108].

Consider a system is operated for a number of times $r_{hist}^1, r_{hist}^2, \ldots, r_{hist}^R$, where the error logs and failure occurrences of these runs are stored in a data repository. Using this run history, we perform the regression in a number of steps. First, we determine the
set of known failure modes \( F = \{ m_0, m_1, \ldots, m_M \} \), where each \( m \in F \) had appeared at least once as failure occurrence in these history runs. Second, for each log record \( n \) in a history run \( r \in \{ r_{\text{hist}}^i \}_{i=1}^R \), we create an error spread-signature \( s_{r,n} \) and derive an estimated eventuality \( h_{r,n}^m \) for each failure mode \( m \in F \) using Equations 5.1 and 5.3, respectively.

Third, we combine the created signatures and estimated failure mode eventualities of all history runs in \( M+1 \) lists: \( L^{\text{sig}} \) and \( \{ L_m^{\text{ev}} \}_{m=1}^M \). Therefore, \( L^{\text{sig}} = \cup^r L^{\text{sig}}_r \) and \( L_m^{\text{ev}} = \cup^r L_m^{\text{ev}}_r \), where \( L_r^{\text{sig}} \) and \( L_r^{\text{ev}} \) are the lists of signatures and failure mode \( m \) eventualities in the run \( r \), respectively. Fourth, we generate \( M+1 \) matrices: \( X^{\text{reg}} \) and \( Y_m^{\text{reg}} \). \( X^{\text{reg}} \) is a \((J+1) \times N\) matrix and \( Y_m^{\text{reg}} \) is a \(1 \times N\) matrix, where \( J \) is the number of signature variables and \( N \) is the number of signatures in \( L^{\text{sig}} \). \( X^{\text{reg}} \) represents the signatures listed in \( L^{\text{sig}} \) and \( Y_m^{\text{reg}} \) represents the failure mode eventualities listed in \( L_m^{\text{ev}} \) of a failure mode \( m \). Therefore, the matrices \( X^{\text{reg}} \) and \( Y_m^{\text{reg}} \) are expressed as follows.

\[
X^{\text{reg}} = \begin{bmatrix}
1 & v_1^1 & v_2^1 & \cdots & v_J^1 & \cdots & v_1^J & \cdots & v_J^J \\
1 & v_2^1 & v_2^2 & \cdots & v_J^2 & \cdots & v_2^J & \cdots & v_J^J \\
\vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
1 & v_n^1 & v_n^2 & \cdots & v_n^J & \cdots & v_n^J & \cdots & v_n^J \\
\vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
1 & v_N^1 & v_N^2 & \cdots & v_N^J & \cdots & v_N^J & \cdots & v_N^J 
\end{bmatrix},
Y_m^{\text{reg}} = \begin{bmatrix}
h_1^m \\
h_2^m \\
\vdots \\
h_n^m \\
\vdots \\
h_N^m 
\end{bmatrix}
\]

(5.5)

In the fifth step, we use the OLS method to correlate \( X^{\text{reg}} \) with each \( Y_m^{\text{reg}} \) to determine an estimator \( \alpha_m \) for each failure mode \( m \). The estimator \( \alpha_m \) is derived by the OLS as a matrix \( b_m \) of \((J+1) \times 1\) coefficients of the signature variables \( 1, v_1, v_2, \ldots, v_J \) as follows.

\[
b_m = (\bar{X}X)^{-1}\bar{X}Y_m
\]

(5.6)

where \( X \) represents \( X^{\text{reg}} \) and \( Y_m \) represents \( Y_m^{\text{reg}} \). \( \bar{X} \) is the matrix transpose of \( X \) and
(\(\bar{X}X\))^{-1} is the inverse matrix of \(\bar{X}\) multiplied by \(X\) from the right. By using Eq 5.6, we can express the estimator as \(\alpha^m(X) = Xb_m\). Therefore, for any signature \(s_n\), we can use \(\alpha^m\) to predict the eventuality of a failure mode \(m\) as follows.

\[
\hat{h}_n^m = X_n b_m
\]  

(5.7)

where \(\hat{h}_n^m\) indicates the predicted failure mode eventuality from signature \(s_n\) represented as one row (vector) matrix \(X_n\) as follows.

\[
X_n = \begin{bmatrix} 1 & \upsilon^1_n & \upsilon^2_n & \ldots & \upsilon^j_n & \ldots & \upsilon^l_n \end{bmatrix}
\]  

(5.8)

5.2.5 Resolving Prediction Results

If a system run tends to fail, some failure mode eventualities are expected to continuously increase and approach a value of 1 towards the end of this system run. If the system run is successful, then all failure mode eventualities are ideally expected to approach 0 throughout the runtime. By using accumulated error counters in log record signatures, each runtime signature provides a clearer vision than all of its previous runtime signatures about a system error state. Therefore, at the end of a system runtime, the predicted failure mode eventualities are ideally expected to comply with the system success or failure state. Therefore, we depend on the last predicted failure mode eventualities of a system runtime to resolve the failure prediction results. By resolving the prediction results, we mean utilizing the predicted failure mode eventualities to estimate two parameters: failure occurrence state \(\hat{f}^o\in\{success, failure\}\) and mode \(\hat{f}^m\in\{m_0, m_1, \ldots\}\) in case of failure.

To obtain these parameters, we define an eventuality threshold \(\tau\in[0,1]\) and use it to control the prediction sensitivity. The choice of \(\tau = 0\) indicates that a predicted
system state is considered to be successful, if and only if the eventualities of all failure modes are equal to 0. The choice of $\tau = 1$ indicates that a predicted system state is considered successful, if and only if the eventualities of all failure modes are less than 1. This threshold can be specified based on expert opinion. In the experimental evaluation section, we choose $\tau = 0.3$ as a reasonable tolerance factor for the prediction approach that avoids false alarms of failures with low predicted eventualities.

We predict a failure occurrence as the system state where any of the failure mode eventualities exceeds the threshold $\tau$. We predict the mode of this failure based on the maximum estimated eventuality among all modes. Let $h_{\text{max}}$ be the maximum eventuality of all modes. Therefore, if $h_{\text{max}} < \tau$, then $\hat{f}^o = \text{success}$, else $\hat{f}^o = \text{failure}$. Finally, if $\hat{f}^o = \text{failure}$, we estimate $\hat{f}^m$ as the failure mode $m$ that has maximum eventuality.

Several measures can be used to evaluate the regression analysis and failure prediction results. The estimators that result in the regression analysis can also be evaluated by the correlation coefficient and regression sum of square errors. Based on an estimator, a correlation coefficient can be derived as an indicator for the degree of dependence between these signatures and the eventualities of failures in the system runs. The regression sum of square error measures the discrepancy between the actual data and the estimation model. Abdi et al. [109] derive the correlation coefficient as follows.

$$
\rho^m = \frac{\bar{b}_m \tilde{X} \tilde{Y}_m - \frac{1}{N} (\tilde{1} \tilde{Y}_m)^2}{\tilde{Y}_m \tilde{Y}_m - \frac{1}{N} (\tilde{1} \tilde{Y}_m)^2} \tag{5.9}
$$

where $X$ is an $N \times (J + 1)$ matrix of the values of $1, x_1, x_2, \ldots, x_J$ in $N$ rows and $Y_m$ is an $N \times 1$ vector matrix of the corresponding observations for the dependent variable. In this equation, $\bar{b}_m$, $\tilde{X}$, and $\tilde{Y}_m$ are the matrix transpose of $b_m$, $X$, and $Y_m$, respectively, and $\tilde{1}$ is a matrix of 1’s that is conformable with the transpose of $Y_m$. The regression sum of square errors can be derived as follows [109].
Based on a failure prediction operation, we derive the prediction accuracy with respect to failure occurrences and failure modes. These prediction accuracies can be evaluated by comparing the actual and the predicted failure results. Let \( \sigma_f \) and \( \sigma_m \) indicate the failure occurrence mode prediction accuracies. To derive these accuracy measures, we use the numbers of true negatives \( T^- \), false negatives \( F^- \), true positives \( T^+ \), and false positives \( F^+ \) with respect to each measure as follows.

\[
\sigma = \frac{T^- + T^+}{T^- + T^+ + F^- + F^+}
\]  

(5.11)

where \( \sigma \) represents either of \( \sigma_f \) or \( \sigma_m \) by considering the prediction results according to \( \hat{f}_o \) or \( \hat{f}_m \) measures, respectively as explained earlier.

5.2.6 Performance Overhead

Two main factors can be responsible for performance degradation in a system due to the use of a failure prediction technique: the instrumentation and the prediction overhead. The software instrumentation increases a system memory utilization and processing overhead due to the instrumented code statements responsible for capturing system execution states. The failure prediction process increases the performance overhead due to the execution of the forecasting functions over the captured execution states. Fortunately, our failure mode estimator does not cause any performance degradation based on these two factors. First, our prediction technique does not rely on any system internal states and it does not require any instrumentation of the software system. The technique utilizes error log records that are available in most of the software systems. Second, our failure prediction process does not introduce any performance overhead to the executed software system.
since this process is operated independent of the system. However, executing the failure prediction process on separate processing environment may introduce other performance overhead due to the message passing (or file access locking) with the executed systems. However, these performance overhead can be considered negligible relative to the other embedded failure prediction techniques based on code instrumentation. However, in our technique, we prepare the failure mode estimators offline from system error history. Each estimator is practically a number of coefficients for the error spread signature variables. Predicting a failure mode eventuality in correspondence to each error log record using these estimators will not cause any message delay or file lock if these log records are generated in relatively wide windows of time. To summarize, our failure prediction does not introduce any considerable performance overhead based on instrumentation or prediction process execution. Minimal overhead may be introduced due to message passing or file locking for relatively small time windows of log record generation.

5.2.7 Regression and Prediction Processes Summary

Our runtime failure prediction approach involves two main steps: regression and prediction. In the regression step, system error log and failure history are used to determine an estimator $a^m$ for each failure mode $m$. The runtime signatures are generated for the log records of these history runs and failure mode eventualities are calculated for each log record using Equations 5.1 and 5.3, respectively. The runtime signatures of the combined error log trails of these runs are used to generate a matrix $X^{reg}$ using Equation 5.5. The failure mode results are used to create a matrix $Y^{reg}_m$ for each failure mode that has occurred in these runs using Equation 5.5. Using $X^{reg}$ and $Y^{reg}_m$, we determine $a^m$ with respect to each failure mode $m$ using Equation 5.6. We also measure the correlation coefficient of these regression functions and the regression sum of square errors using Equations 5.9 and 5.10, respectively.
In the prediction step, while the system is executed, failure modes are predicted from the error log trails using the estimators \( \{\alpha^m\}_{m=1}^M \). For each runtime error log record \( n \), we determine an error spread signature using Equation 5.1 and create a vector matrix \( X^n \) using Equation 5.8. We use this matrix with each estimator \( \alpha^m \) to predict the eventuality of each failure mode using Equation 5.7. Finally, the predicted failure occurrence and mode are resolved from the eventualities of all predicted failure modes as explained in Section 5.2.5. In the following paragraph, we illustrate the types of log trails and failure modes relative to a set of estimators during this regression and prediction process.

The estimators \( \{\alpha^m\}_{m=1}^M \) are derived for failure modes that have occurred during a system run history. Therefore, relative to this set, an error log trail can be old or new and a failure mode can be known or unknown. An old log trail is a number of log records that is used as part of the log history to derive the set of estimators. A known failure mode is a failure mode that has appeared in any of the runs used to derive the set of estimators. A new log trail and an unknown failure mode are the opposites of an old log trail and a known failure mode, respectively. In our experiment, we use these types of log trails and failure modes to examine the failure prediction technique and determine its accuracy with respect to these log trails and failure mode types.

5.3 Experimental Evaluation

We evaluate the failure mode prediction technique using the PostgreSQL 8.4.4 database software system. We aim to measure the accuracy of our method with respect to failure occurrence and mode predictions. We use the same experimental environment as explained in Subsection 3.7.1. In this section, we describe our experimental process and setup. We present the actual and predicted failure occurrences and modes in the system runs. We show the changes in the predicted failure mode eventualities throughout some example system runs. Finally, we compare our failure prediction results to the actual failure
5.3. EXPERIMENTAL EVALUATION

occurrences and modes and derive their prediction accuracies in three categories of log trail and mode types.

5.3.1 Experimental Process and Setup

Through the experimental process, we run the PostgreSQL database server for a number of times, consider them as history runs, combine their error log data, and use this data to determine the failure mode estimators. We use the estimators to predict the failure mode occurrences in these runs as old log trails. We further run the database server some more times as new log trails and predict the failure modes during these runs using the determined estimators. The whole experimental process is repeated for 10 times using 10 different experimental data sets. PostgreSQL 8.4.4. is assumed to be a fault free version.

To enforce a probability of runtime failures and diversify the possible failure modes, we generate 10 versions by altering random program statements and use them in the different system runs. In the experimental process, we examine the approach by executing these versions against 10 different experimental data sets (see Subsection 4.4.1).

<table>
<thead>
<tr>
<th>Regression</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_0 )</td>
<td>( v_0 )</td>
</tr>
<tr>
<td>( E_1 )</td>
<td>( r_{1,0} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( E_9 )</td>
<td>( r_{9,0} )</td>
</tr>
</tbody>
</table>

Table 5.3: The system runs utilized in the regression and prediction steps, their experimental data sets, and utilized system versions.

The experimental process involves regression and prediction steps as explained in Section 5.2.7. Table 5.3 lists the system runs used for regression and prediction steps of our experimental process. The table columns are the system runs and the table rows are the experimental data sets. Thus, each table row describes an individual experiment.
using a specific experimental data. We refer to experiment $i$ as $E_i$ and we refer to the system runs in this experiment as \{\(r_{i,0}, r_{i,1}, \ldots, r_{i,9}\)\} for run 0, run 1, \ldots, run 9, respectively.

We capture the system standard output streams of system runs to determine the success/failure states of each run and observe the failure mode in case of system failure. The start and end times of the system execution are exported for each run in a performance log file. The system output was directed from the standard output stream to text files in order to allow comparing the final results of the system runs. The system performance was measured in microseconds from the start and end times. The execution time of these runs were compared to the runtime of version 0 with respect to the same experimental data sets to identify the early and late service delivery failures. Similarly, the outputs of the system runs were compared to version 0 (original version) of the software to identify the correctness of the system results.

In the regression step, the data set is utilized to operate the system for 5 times \(r_{i,0}, r_{i,1}, \ldots, r_{i,4}\) using versions \(v_0, v_1, \ldots, v_4\), respectively. These runs are used as system history to determine an estimator \(\alpha^m_i\) for the \(i\)-th experiment with respect to each failure mode \(m\). In the prediction step, the system is executed for 10 times \(r_{i,0}, r_{i,1}, \ldots, r_{i,9}\) using versions \(v_0, v_1, \ldots, v_9\). Failure modes are predicted from the error log trails during these runs using the estimators \(\{\alpha^m_i\}_{m=1}^{M} : i \in \{1, 2, \ldots, 10\}\).

Finally, we calculate the prediction results and compare them to the actual system failure behavior reported in the system output stream. Based on the prediction results and comparison to the actual failure mode occurrences, we derive the prediction accuracy of failure occurrences and modes using the old log trail from the system history (runs 0 to 4) and using the new log trail (runs 5 to 9) with both known and unknown failure modes according to their occurrences in the history runs.
5.3. EXPERIMENTAL EVALUATION

5.3.2 Actual Failure Occurrences and Modes in System Runs

Now, we demonstrate the findings of this experimental evaluation. By executing the 10 software system versions against the 10 experimental data sets, we obtained a number of failure modes as listed in Table 5.4. In these system runs, eleven failure modes have occurred. Most of the failure modes \((m_1, m_2, \ldots, m_8)\) were already written to the system standard output stream and the system stopped working after their occurrences. The other three failure modes \((m_0, m_9, \text{ and } m_{10})\) were observed manually. Failure mode \(m_0\) indicates a deviation of the database query result from the expected result. This failure is detected by comparing the system results of each run with the results of run 0. Note that, run 0 of each experiment uses version 0 which has no injected faults. Failure modes \(m_9\) and \(m_{10}\) indicate performance deviation (late and early, respectively) relative to the expected run time of the system with respect to each experimental data set. Similar to \(m_0\), we compare the time of each run \(r\) with run 0 using the same experimental data set to measure the time deviation of this run. By using the actual execution times of system runs in Table A.1, we identify the performance failure states of these runs. We consider only the time deviations greater than 60 second as performance failures. The choice of 60 seconds helps decrease the number of performance failure occurrences considered for the regression analysis procedure.

| \(m_0\) | Incorrect query results. |
| \(m_1\) | Server process (PID) was terminated by signal 6: Aborted, failed assertion (dll_tail). |
| \(m_2\) | Incorrect SQL statement: syntax error at or near “SQL command” at character “#”. |
| \(m_3\) | Server process (PID) was terminated by signal 6: Aborted, failed assertion (argument). |
| \(m_4\) | Server process (PID) was terminated by signal 6: Aborted, failed assertion (refcount). |
| \(m_5\) | Server process (PID) was terminated by signal 6: Aborted, failed assertion (mergeclause_list). |
| \(m_6\) | Server process (PID) was terminated by signal 6: Aborted. |
| \(m_7\) | Server process (PID) was terminated by signal 9: Killed. |
| \(m_8\) | Cannot drop table (TBL) because other objects depend on it. |
| \(m_9\) | Server process (PID) was terminated by signal 11: Segmentation fault. |
| \(m_{10}\) | Late service delivery. |
| \(m_{10}\) | Early service delivery. |

Table 5.4: Description of the failure modes that have occurred in the systems runs.
5.3. EXPERIMENTAL EVALUATION

Table 5.5: The actual failure mode occurrences in systems runs.

<table>
<thead>
<tr>
<th>Run 0</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_0</td>
<td></td>
<td>m_1</td>
<td>m_3</td>
<td></td>
</tr>
<tr>
<td>E_1</td>
<td></td>
<td>m_1</td>
<td>m_3</td>
<td>m_5</td>
</tr>
<tr>
<td>E_2</td>
<td>m_5</td>
<td>m_5</td>
<td>m_5</td>
<td></td>
</tr>
<tr>
<td>E_3</td>
<td>m_4</td>
<td>m_5</td>
<td></td>
<td>m_8</td>
</tr>
<tr>
<td>E_4</td>
<td>m_4</td>
<td>m_6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E_5</td>
<td>m_4</td>
<td>m_5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E_6</td>
<td></td>
<td>m_5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E_7</td>
<td></td>
<td>m_4</td>
<td>m_5</td>
<td></td>
</tr>
<tr>
<td>E_8</td>
<td>m_4</td>
<td>m_7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E_9</td>
<td>m_10</td>
<td>m_10</td>
<td>m_9</td>
<td></td>
</tr>
</tbody>
</table>

Some of these failures have occurred only one time and other have occurred more than once. Table 5.5 displays the actual failure mode occurrences in the system runs in each experiment. The table rows are the experiments and the columns are the system runs of each experiment. In the table, empty cells indicate successful system runs, otherwise, the cell displays the failure mode of this run. Since version 0 is not injected with faults, the first run was successful for all experiments. Some failure modes have occurred only once (e.g., m_0 and m_7), while the other failure modes have occurred repeatedly across the experiments and the runs. For example, failure mode m_4 is repeated across experiments 3, 4, 5, 7, and 8. Failure mode m_10 is repeated across runs 1, 2, and 4-8. By considering the first 5 runs as history in the regression step, failure modes in these runs will be known to the estimator. Therefore, m_2 and m_8 in runs 7 and 9, respectively are unknown failure modes, since they have not occurred in the history runs.

5.3.3 Example Error Log Trails

To illustrate the change of the error spread-signatures during system runtime, we show some examples from the actual experimental evaluation. Figure 5.3 shows the distributions of signature variables with respect to four different system runs. Of these four, we selected two successful (r_2,2 and r_2,8) and two failure (r_7,3 and r_8,4) runs. The failure
modes of $r_{7,3}$ and $r_{8,4}$ are $m_4$ and $m_8$, respectively. In these figures, the X-axis is the error log record points and the Y-axis is the number of errors. Each graph represents a signature variable which represents an error state pattern. We notice the following observations with respect to the change of signature variables along system runtime. First, signature variables change incrementally or stay unchanged along time intervals of system runs. The increase indicates error occurrences during the error log interval of the specific log record. Second, occasionally, some variables are constant and equal to 0 indicating error free states with respect to specific error patterns. Third, the signature variables with higher values differ among system runs. This difference indicates that in some system runs some error state patterns may occur most often relative to other error state patterns.
5.3. EXPERIMENTAL EVALUATION

5.3.4 Correlation and Prediction Error Measures

We present our regression analysis results by listing the degrees of correlation between the regression matrices and the output matrices. Figure 5.4 presents the resulting correlation coefficients of the regression analysis step of our experiments. These coefficients describe the degrees of dependence between the failure mode results and the system error log record of each run. In Figure 5.4, the X-axis lists the failure modes, the Y-axis lists the correlation coefficients, and the Z-axis (the depth) lists the 10 experiments. We first notice that the average correlation coefficient of $E_0$ is 0. The reason for that is the non-occurrence of any failure during the first 5 runs of this experiment. We also notice that, most correlation coefficients are equal to 0. This is due to the occurrence of only few failures in the first 5 runs of each experiment. For example, experiment $E_9$ has 0 correlation coefficient with all failure modes except for $m_9$ and $m_{10}$.

<table>
<thead>
<tr>
<th></th>
<th>$E_0$</th>
<th>$E_1$</th>
<th>$E_2$</th>
<th>$E_3$</th>
<th>$E_4$</th>
<th>$E_5$</th>
<th>$E_6$</th>
<th>$E_7$</th>
<th>$E_8$</th>
<th>$E_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correlation coefficient</td>
<td>-</td>
<td>0.991</td>
<td>0.629</td>
<td>0.990</td>
<td>0.621</td>
<td>0.576</td>
<td>0.999</td>
<td>0.993</td>
<td>0.893</td>
<td>0.999</td>
</tr>
<tr>
<td>Regression square error</td>
<td>-</td>
<td>0.015</td>
<td>0.278</td>
<td>0.008</td>
<td>0.296</td>
<td>0.309</td>
<td>0.001</td>
<td>0.005</td>
<td>0.112</td>
<td>0.050</td>
</tr>
</tbody>
</table>

Table 5.6: The correlation coefficients and the regression square errors averaged among the failure mode prediction functions.
5.3. EXPERIMENTAL EVALUATION

From Table 5.5, we notice that these two failure modes \((m_9\) and \(m_{10}\)) particularly have occurred in the first five runs of experiment \(E_9\). Table 5.6 lists the average nonzero correlation coefficients and average regression square errors among all failure modes for the 10 experiments. The second row of the table lists the regression square errors. Similar to the correlation coefficient of \(E_0\), the regression square error is not calculated since no errors or failures have occurred. From the table, except for \(E_2\), \(E_4\), and \(E_5\), the correlation coefficients of all experiments are notably high. This indicates that, the regression function can efficiently predict these correlated failure modes in all experiments. However, to explain the reasons that \(E_2\), \(E_4\), and \(E_5\) have relatively low correlation coefficients, we find the similarities of the log variable distributions in these particular three experiments and compare these similarities to the remaining experiments. Thus, we observe three characteristics of the error log trails of these particular experiments. These characteristics are as follows: higher number of errors in the error log trail of run 2 relative to runs 3 and 4; higher number of records in the error log trail of run 2 relative to runs 3 and 4; and run 2 was successful while runs 3 and 4 involved failures. These characteristics of the error log trails could occur in the system runs where incorrect control transitions are numerous and ineffective (e.g., repeating simple assignment statement or deleting a print statement within a loop). However, this case affects the run history data used for the regression of these three experiments by combining two types of log trails: trails with numerous log records, high numbers of errors, and the successful run results in log trails with fewer log record numbers, fewer number of errors, and failure mode occurrences. Obviously, this inharmonious log data combination is also the cause of relatively high regression square errors for these experiments \((E_2, E_4, \text{ and } E_5)\).
5.3. EXPERIMENTAL EVALUATION

5.3.5 Predicted Failure Mode Eventualities Along System Runtime

To observe the failure mode prediction behavior during system runtime, we present some example system success and failure runs with old and new log trails, and correct and incorrect prediction results. These examples help illustrate the change in the predicted failure mode eventualities during system runs.

Figure 5.5 presents four example eventuality relationships with error spread-signatures in correct attempts of failure mode prediction. These examples involve two failure runs ($r_{1,2}$ and $r_{8,3}$) with modes $m_1$ and $m_4$, respectively and two successful runs ($r_{1,4}$ and $r_{8,2}$). The failure results of these system runs are predicted using old log trails. In this figure, the X-axis is the log record number, the Y-axis is the failure mode eventuality measure, and the Z-axis lists the known failure modes. In Figure 5.5(a), we notice that, only the eventuality of failure mode $m_1$ is increasing along system runtime while the remaining failure mode eventualities have approximately 0 values. By reviewing this run...
5.3. EXPERIMENTAL EVALUATION

(a) Failure mode $m_{10}$ in run 8 of $E_9$

(b) Failure mode $m_{10}$ in run 7 of $E_9$

(c) Successful run 8 of $E_1$

(d) Successful run 8 of $E_2$

Figure 5.6: Example correct and incorrect predictions of success and failure runs with new log trails.

$r_{1,2}$ in Table 5.5, we notice that the predicted failure mode is similar to the actual mode occurrence in the table. Similarly, in Figure 5.5(b), the predicted failure mode of run $r_{8,3}$ is similar to the actual occurrence. In the two runs, the failure mode eventualities are both approaching to 1 with values 0.9957 and 0.9091 at the end of the runs. In Figure 5.5(c), the run $r_{1,4}$ is successful. In this figure, we notice that, the maximum eventuality of all the failure modes did not exceed 0.02 along the system runtime interval. Figure 5.5(d) is similar to Figure 5.5(c), while the predicted eventuality is relatively high. However, this eventuality is still less than the threshold $\tau = 0.3$.

Figure 5.6 presents failure prediction attempts of system runs with new log trails. In this figure, we provide two runs ($r_{9,8}$ and $r_{9,7}$) with failure mode $m_{10}$ and two successful runs ($r_{1,8}$ and $r_{2,8}$). Figure 5.6(a) displays a correct failure mode prediction of run $r_{9,8}$ with high eventuality. Figure 5.6(b) displays a system run $r_{9,8}$ where the failure mode is correctly predicted with low eventuality. Although the actual failure mode is $m_{10}$ and
5.3. EXPERIMENTAL EVALUATION

that the eventuality of failure mode $m_{10}$ is higher than 0, this prediction result may not be considered correct with the eventuality threshold $\tau$ selected earlier ($\tau = 0.3$). In Figure 5.6(c), although during the system runtime, the eventuality of $m_3$ has increased at the beginning of the runtime, it decreased again towards the end of the runtime interval. Therefore, this system run is predicted as successful. This prediction result complies with the actual occurrence according to Table 5.5. Figure 5.6(d) shows an incorrect prediction result of run $r_{2.8}$. In this attempt, $m_3$ was predicted with eventuality measure of 0.4, while the run is actually successful.

Figure 5.7 presents some example incorrect failure mode prediction results. In Figures 5.7(a) and 5.7(b), incorrect failure modes were predicted with high eventualities in runs $r_{1.9}$ and $r_{4.7}$. The actual failure modes of these runs are $m_8$ and $m_2$, while the predicted modes are $m_3$ and $m_5$, respectively. This incorrect prediction is due to the actual occurrence of unknown failure modes ($m_8$ and $m_2$). The prediction function classifies the failure state to the known failure modes that have similar relationships to the error log.
5.3. EXPERIMENTAL EVALUATION

variable distributions. In Figures 5.7(c) and 5.7(d), while the predicted failure matches the actual occurrence, these prediction attempts are considered as incorrect since the failure mode eventuality measures are low relative to the threshold value.

### 5.3.6 Failure Mode Prediction in System Runs

In this subsection, we list the resolved failure mode results in all system runs in Table 5.7. The table lists the failure mode prediction results of the system runs in all the experiments. The results are displayed in two table parts (left and right) for old and new log trails of runs 0 to 4 and runs 5 to 9, respectively. For each system run, the table displays two measures: maximum predicted eventuality $h_{\text{max}}$ and predicted failure mode $\hat{f}_m$. Empty table cells indicate that the eventualities of all failure modes are equal to 0, i.e., a system run success state is predicted. Using the threshold $\tau$, we can decide about the success or failure states of the remaining run results. However, we notice that the predicted failure mode eventualities vary between 0.0026 and 1. By crosschecking Table 5.7 with Table 5.5 for the predicted and the actual failure mode occurrences, we evaluate the accuracy of our prediction method. For example, the predicted failure mode of run $r_{1,2}$ is $m_1$ with

<table>
<thead>
<tr>
<th>Old error log trail</th>
<th>New error log trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>run 0</td>
<td>run 1</td>
</tr>
<tr>
<td>$h_{\text{max}}$</td>
<td>$f_m$</td>
</tr>
<tr>
<td>$E_0$</td>
<td>0.014 3</td>
</tr>
<tr>
<td>$E_1$</td>
<td>0.070 4</td>
</tr>
<tr>
<td>$E_5$</td>
<td>0.003 10</td>
</tr>
</tbody>
</table>

Table 5.7: The failure mode eventualities of system runs with respect to old and new error log trails.
5.3. EXPERIMENTAL EVALUATION

5.3.7 Comparing Actual and Predicted Failure Results

We determine the accuracy measures of failure occurrence prediction $\sigma_f$ and failure mode prediction $\sigma_m$ as explained in Section 5.2.5. We determine these measures for three categories of system runs based on the type of error log trails (old and new) and actual failure modes (known and unknown). These categories are system runs with old log trails, new log trails and known failure modes, and new log trails and unknown failure modes. Note that, for the last category, we determine only the true and false positive accuracy measures. This is because a successful run state is known to the estimator and thus, it cannot belong to this category of unknown failure modes.

Table 5.8 summarizes the prediction accuracy measures of all the provided experiments split into three parts with respect to the three mentioned categories. The table provides the counts and the percentages of the positive and the negative predictions of failure occurrences and modes. For each of these two prediction measures, we display the numbers and the percentages of the runs of each accuracy measure. Column 3 presents the total number of actual negative results ($-ve$) for old error log trails. Columns 4 and 5 present the true negatives and false negatives, respectively of this category. Columns 6-8 present similar results regarding the positive predictions of the same category. The same pattern
is repeated for columns 9-14 with respect to the new log trails and known failure modes. Finally, columns 16-18 display the accuracy measures of the positive predictions with respect to the third category of new log trails and unknown failure modes. A table cell describes the number of runs classified with an accuracy measure and category. Since a system success state is not associated with any failure modes, unknown modes do not affect the true or false negative measures. Consequently, these measures are shown in the table as “not applicable (N/A)”.

For example, the table shows the accuracy of failure prediction measures of system runs with old error log trails as 83.33% ($T^+$) and 16.67% ($F^+$), respectively. The percentage of $T^+$ is calculated by dividing the number of positive failure runs predicted correctly ($T^+$) and the total number of positives failure runs (+ve) ($i.e., \frac{15}{18} = 83.33\%$). Similarly, the percentage of $F^+$ is calculated by dividing the number of positive failure runs predicted incorrectly ($F^+$) and the total number of positive failure runs (+ve) ($i.e., \frac{3}{18} = 16.67\%$).

It is clear from the table that the percentages of true negatives and positives ($T^-$ and $T^+$) are significantly higher than the false negatives and positives ($F^-$ and $F^+$). The percentages of true negatives of both failure occurrence and mode predictions are equal for old and new log trails, respectively, and their values are 93.75% and 86.21%. The reason for this similarity is that in a true negative prediction, all failure mode eventualities are always less than the threshold $\tau$ and always no failure mode is predicted. The percentages of true positives of failure occurrence and mode predictions are 83.33% and 94.44% using the old log trail, 60% and 100% using the new log trail with known failure modes, and 81.25% and 0% using the new log trail with unknown failure modes. These measures indicate that, the technique forecasts the known failure modes more effectively than the failure occurrences. Otherwise, if the failure modes are unknown, the technique incorrectly associates it with one of the known modes.

According to the accuracy measures of Table 5.8, we present the prediction accuracy
5.4. SUMMARY

<table>
<thead>
<tr>
<th>Error log trail/failure mode</th>
<th>Old/known</th>
<th>New/known</th>
<th>New/unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure occurrence prediction accuracy: $\sigma^s$</td>
<td>90%</td>
<td>82.40%</td>
<td>81.25%</td>
</tr>
<tr>
<td>Failure Mode prediction accuracy: $\sigma^m$</td>
<td>94%</td>
<td>88.24%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5.9: Prediction accuracy of failure occurrence and mode.

for failure occurrence and mode with respect to the same three categories in Table 5.9. It is clear from the table that, the estimator can be used to predict failures and failure modes with high accuracy (90%) for the old log trails and known failure modes. The failure occurrence prediction accuracy is lower when the error log trail is new and the failure modes are known to the estimator, however, it is still reasonably high. When the error log trail is new and failure modes are unknown, the accuracy of the failure occurrence prediction is still high (81.25%). This indicates the ability of the estimator to distinguish failure occurrences even if the failure mode itself is unknown to the estimator. Regarding the failure mode occurrence, the estimators provide 94% and 88.24% prediction accuracies for old error log trails and new error log trails, respectively with known failure modes. Therefore, the estimator is capable with a high accuracy to distinguish the mode of failure in these two run categories. Obviously from the table, despite its ability to predict the occurrence of unknown failure modes, the estimator does not differentiate between these unknown failure modes as already expected.

5.4 Summary

Predicting runtime failures in an early stage of their manifestation is important to assure system resilience and to enable autonomic systems to perform defensive actions prior to a failure occurrence. Most existing failure prediction techniques do not estimate failures during runtime nor they predict failure modes. Moreover, these techniques require extensive instrumentation of the source code. In this chapter, we provide an approach for failure mode prediction during system runtime. Our methodology utilizes system error
log records to craft error spread-signatures that summarize error log variables throughout the system runtime interval. Using system error log history, we determine an estimator between these signatures and failure modes. This estimator is then used to predict a failure mode eventuality (probability of mode occurrence) from system error log variables during system runtime.

Our experimental evaluation results show high accuracy of failure occurrence and mode predictions with respect to old and new log trails. Known failure modes (previously occurred in the system) are predicted and distinguished with high accuracies that exceed 88% for new log trails and 94% for familiar log trails. The failure occurrence of unknown modes are predicted with high accuracy that exceeds 81%, while obviously, these modes are not distinguished since the estimator does not recognize their manifestation manner. The main contributions of this work involve the prediction of failure eventuality during runtime and forecasting the possible failure modes.
Chapter 6

Conclusions, Limitations, and Future Work

This thesis presents a number of techniques that facilitate failure prediction from runtime errors in large software systems. In this chapter, we provide our conclusions, limitations, and future work.

6.1 Conclusions

The proposed techniques in this thesis include control flow representation, control flow monitor, and failure prediction. In the following subsections, we provide our conclusions about these techniques.

6.1.1 Control Flow Representation

Contemporary software systems provide sophisticated functionalities composed through intricate control flow structures. Large program size and structure intricacy are critical challenges for control flow representations for the reliability analysis of these programs. Three main CFG structures are distinguished in current reliability analysis work: Statement-based (S-CFG), Block-based (B-CFG), and Component-based (C-CFG) control flow graphs. These CFGs are either remarkably complex or lack accuracy by allowing some illegal control flow sequences in their representations. The C-CFG is highly abstract
and it only represents inter-component control flow transitions. It is also vulnerable to illegal control flow sequences due to the ignorance of the dependencies among software connections. The S-CFG and B-CFG structures are extremely complex and are not suitable for large software systems.

We propose a Connection Dependence Graph (CDG) for simple representation that considers both inter-component and intra-component control flow transitions. A CDG is a connection-based control flow graph representation of all possible connection invocation sequences that may occur during a program execution. From the program source code, we obtain the software architecture by using our reverse architecting toolkit which derives the software architectures of C-based programs. Based on this software architecture, we describe a number of architectural patterns and exploit them to capture the control transitions. Then, we categorize these transitions into a number of transition groups which are used to identify the execution paths among connections and utilize them to construct the CDG.

We provide an experimental evaluation to examine the effect of program size on the CDG and the other CFGs by comparing them with respect to complexity using the PostgreSQL 8.4.4 open source database software system. Our results show that, the CDG has relatively small number of nodes and edges in comparison to the S-CFG and the B-CFG. Regarding representation inaccuracy, the CDG always has lower number of illegal branches than both B-CFG and C-CFG. Moreover, it shows that, the CDG mostly has fewer illegal branches than S-CFG, especially with respect to large program sizes. The main advantages of the CDG involve avoiding the complexities and inaccuracies of the existing CFG structures. The CDG is not limited to the inter-component control transitions and its representation outlines comprehensive details about the intra-component control flow transitions. Moreover, the CDG preserves the connection invocation dependencies among software components. Our CDG derivation algorithm can also be used to derive
6.1. CONCLUSIONS

the other CFG structures.

6.1.2 Control Flow Monitor

Control flow errors are major impairments to software system correctness. Several run-
time monitoring approaches were proposed to detect control flow errors by using watch-
dog, redundancy, assertion, or monitoring approaches. The major limitations in these
techniques involve high performance overhead, intensive program modifications, and use
of extra hardware. Signature-based control flow monitors can save more performance
overhead than the other techniques and at the same time they do not require any addi-
tional hardware or software redundancy. However, existing signature-based control flow
monitoring approaches rely on the B-CFG to represent the system control flow structure.
Therefore, in these techniques, validating the control flow transitions may not be feasible
for large software systems and may be susceptible to false negatives due to the inclusion
of these illegal control flow branches in the B-CFG. The utilization of the CDG in our
monitor decreases the possibility of false negatives due to its higher accuracy relative to
the B-CFG. Moreover, our connection-based signature structure allows complete coverage
of all program connections and it can determine the control flow errors with respect to
the unreturned connections in case of improper program termination.

We propose a runtime monitoring approach for control flow error detection using
connection-based signatures. This signature structure covers more program connections
and associates each one to a unique signature. We describe the connection-based signa-
ture structure, explain how to partition a program into CIBs, and associate a CDG to
each partition. We provide our control flow monitor structure and checking algorithm
based on these CDGs of program partitions. An experimental evaluation of the proposed
approach is provided using PostgreSQL open-source database. The results show that our
6.1. CONCLUSIONS

Technique is capable of detecting control flow errors in 10 different versions with variable incremental numbers of randomly altered program statements with respect to the software control flow structures. The detected control flow errors in these versions are found to be mostly increasing with the number of altered program statements. Our control flow monitor is able to detect the improper system termination by distinguishing the unreturned connection invocations. Regarding the error state parameters, most of the detected control flow errors are illegal branches of external transitions and they are also not repeated in previous system states.

Our signature structure for control flow error detection helps cover software connections that could not be considered in the existing techniques due to the limitations of branch-free block partitioning. It also enables tracking the connection invocations and their return addresses and allows relating a runtime state to the execution traces of a software system. The execution tracking helps determine important error parameters (e.g., unreturned control transitions). Furthermore, unlike existing techniques, the usage of the CDG allows injecting only one line of code with respect to each graph node. Given that, our control flow monitor saves half of the performance overhead in comparison to the existing techniques that inject two lines in each block. Our connection-based signature structure helps locate the responsible component of each control flow error. It further narrows down the location to the code statement at which an illegal branch or unreturned connection has taken place.

6.1.3 Failure Prediction

Leading software manufacturers and industrial organizations use failure prediction methods to minimize the failure risks and possible consequences. Predicting runtime failures in an early stage of their manifestation is important to enable autonomic systems to perform defensive actions prior to failure occurrence. Most existing failure prediction techniques
do not forecast failures during system runtime. Instead, they forecast failure numbers and estimate the effects of possible failure modes that may occur in future time intervals. Existing runtime failure prediction techniques do not estimate the failure modes and do not predict the eventuality of failure occurrence. They also require extensive instrumentation of source code.

We provide an approach for failure mode prediction during system runtime. Our methodology utilizes system error log records to craft error spread-signatures that summarize error log variables throughout the system runtime interval. Using system error log history, we determine an estimator between these signatures and each failure mode. This estimator is then used to predict a failure mode eventuality (probability of mode occurrence) from system error log variables during system runtime. Our experimental evaluation results show high accuracy of both failure predictions. Known failure modes (previously occurred in the system) are predicted and distinguished with high accuracies that exceeds 88% for new log trails and 94% for familiar log trails. The failure occurrences of unknown modes are predicted with high accuracies that exceeds 81%, while obviously, these modes are not distinguished since the estimator does not recognize their manifestation manner.

The main contributions of this work involve the prediction of failure eventuality during runtime and forecasting the possible failure modes in the early stages of their manifestation. The technique can be used by self-configuring and self-healing systems to achieve system resilience and avoid failure consequences. This research enables the practical consideration of different criticality levels or severities of different failure modes in the reliability analysis.
6.2 Limitations and Future Work

In the following subsections, we present the limitations of the proposed techniques and the future work derived from these limitations.

6.2.1 Control Flow Representation

The CDG mainly suits the procedural programming paradigm and event-driven software architectures. Further considerations are needed to allow representing software systems of other programming paradigms (e.g., object-oriented programming). In future, we will investigate the required annotations to implement this graph in various programming paradigms. We also aim to further simplify the CDG and enhance its accuracy by implementing a transitive reduction mechanism that divides it into smaller CDGs associated with the CIBs of a software program. The reduced structure will also simplify the procedure call tracking and the search algorithms among the possible execution state trees. We also plan to investigate the impact of using the CDG rather than the existing CFG structures in the architecture-based reliability assessment of software systems.

6.2.2 Control Flow Monitor

Although our control flow monitor require less code instrumentation than the existing techniques, we believe that, exporting a runtime signature prior to each call statement may still cause considerable overhead. We aim to alter our code instrumentation method to patch the process of exporting and validating the system runtime signatures with minimal time delays. Our future work involves exploiting the error state parameters to predict the possible failure occurrences and their types (value, silent, performance, etc). It also involves exploiting these parameters for diagnosing faults and predicting their repairs. A confidence factor based on these parameters will be used with our prediction. Other future work can be classifying components using the correlation of the detected errors
with the system performance. This classification helps determine plausible component crashes according to the history of its errors. We also aim to study the relationships (if any) between data flow errors and control flow error transitions. This relationship can later be used to construct a hybrid control and data flow approach with minimal amount of redundancy.

6.2.3 Failure Prediction

The main limitation of this technique is that it attempts to associate an unknown failure mode with the previous failure mode occurrences, where precisely, it should be identified as an “unknown” failure. The other limitation is evaluating the prediction method according to the assumption that, a system failure state is characterized by only one failure mode. In our future work, we plan to overcome these limitations. We will also involve predicting the time to fail and the time to start a corrective action. In our technique, the failure mode eventualities are calculated at the system level. Other extension will involve determining them at the component levels. Our future work also involves investigating the methodology to decide about the value of the prediction threshold $\tau$ and to study its effects on the implementations of some corrective actions.
Bibliography


Appendix A

Sample Inputs/Outputs of the Proposed Techniques

In this appendix, we mainly present some artifacts that are used throughout the experimental evaluations of the proposed techniques.

A.1 XML Representation

We present the XML document headers used by our reverse architecting toolkit and provide some samples of our architectural representation of the PostgreSQL database server as explained in Subsection 3.5.4.

A.1.1 XML Headers

Figure A.1 describes the XML document structure used to represent the software architecture. The figure shows the header file used during the XML document generation.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE Components [ 
3  <!ELEMENT Components (Component*)> 
4  <!ATTLIST Components 
5      Num_Components CDATA #REQUIRED 
6      Main_folder CDATA #REQUIRED 
7      Include_folder CDATA #REQUIRED 
8      Arch_XML_file_folder CDATA #REQUIRED>
9  <!ELEMENT Component EMPTY> 
10  <!ATTLIST Component 
11      Id CDATA #REQUIRED 
12      Name CDATA #REQUIRED 
13      Num_CIBs CDATA #REQUIRED 
14      Program_File CDATA #REQUIRED 
15      Arch_XML_file CDATA #REQUIRED>
16 ]>
```

Figure A.1: The header file of the XML document representing the program structure.
Similarly, Figure A.2 describes the XML document structure used to represent a software component architecture. The figure shows the header file used during the generation of the XML document representing each component.

```xml
<?xml version = "1.0"?>
<!DOCTYPE Component [ 
<!ELEMENT Component (CIB*)>
<!ATTLIST Component Id CDATA #REQUIRED Name CDATA #REQUIRED Num_CIBs CDATA #REQUIRED>
<!ELEMENT CIB (Provide|Require|Stat_block|EMPTY)>
<!ATTLIST CIB Id CDATA #REQUIRED Name CDATA #REQUIRED>
<!ELEMENT Provide (Call_invoke_Id|Variable Access_Id|EMPTY)>
<!ATTLIST Provide Num_call_invokes CDATA #REQUIRED Num_variable_access_times CDATA #REQUIRED>
<!ELEMENT Require (Call_invoke_Id|Variable Access_Id|EMPTY)>
<!ATTLIST Require Num_call_invokes CDATA #REQUIRED Num_variable_access_times CDATA #REQUIRED>
<!ELEMENT Call_invoke_Id EMPTY>
<!ELEMENT Variable Access_Id EMPTY>
<!ELEMENT Stat_block (Selection|Condition|Loop|Expression)>
<!ATTLIST Stat_block Num_statements CDATA #REQUIRED Braced CDATA #REQUIRED File_location CDATA #REQUIRED>
<!ELEMENT Selection (Stat_block|EMPTY)>
<!ATTLIST Selection It_blocks CDATA #REQUIRED File_location CDATA #REQUIRED>
<!ELEMENT Condition (Stat_block|EMPTY)>
<!ATTLIST Condition File_location CDATA #REQUIRED>
<!ELEMENT Loop (Stat_block|EMPTY)>
<!ATTLIST Loop Id CDATA #REQUIRED File_location CDATA #REQUIRED>
<!ELEMENT Expression (Call|Variable|EMPTY)>
<!ATTLIST Expression Data_access_statement_type CDATA #REQUIRED File_location CDATA #REQUIRED>
<!ELEMENT Call (Expression|EMPTY)>
<!ATTLIST Call Id CDATA #REQUIRED Name CDATA #REQUIRED Collect CDATA #REQUIRED Access_CIB CDATA #REQUIRED Home_CIB CDATA #REQUIRED>
<!ELEMENT Variable EMPTY>
<!ATTLIST Variable Id CDATA #REQUIRED Name CDATA #REQUIRED Access_type CDATA #REQUIRED Access_CIB CDATA #REQUIRED Home_CIB CDATA #REQUIRED>
]>
```

Figure A.2: The header file of the XML document representing the program components.
A.1.2 XML Documents

Some samples of the generated XML documents using our reverse engineering toolkit are shown in this subsection. The toolkit generates two types of XML documents: software and component structure. Here, we show a sample of each type.

Figure A.3 displays a software structure XML document. The document presents the structure of the PostgreSQL version 8.4.4 database system. The document lists 1,178 components representing the “.c” and “.h” files in the database software project.

```xml
<?xml version="1.0"?>
<DOCTYPE Components PUBLIC "-//W3C//DTD Component Index Version 1.0//EN" "Components.dtd">
<Components Arch_XML_file_folder="C:\Atel\990-ExpData\01-ArchProfile\" Include_folder="C:\Atel\990-ExpData\00-SourceCode\PostgreSQL 8.4.4\" src="PostgreSQL 8.4.4\" src="Num_Component="1178">
  <Component Arch_XML_file="comp-0-backend.main.main.c.xml">
    <Program_File>$MAINS\backend\main\main.c Num_CIBs="6" Name="backend.main.main.c" Id="0"/>
  </Component>
  <Component Arch_XML_file="comp-1-backend.access.common.heatmap.c.xml">
    <Program_File>$MAINS\backend\access\common\heatmap.c Num_CIBs="24" Name="backend.access.common.heatmap.c" Id="1"/>
  </Component>
  <Component Arch_XML_file="comp-2-backend.access.common.indextemplate.c.xml">
    <Program_File>$MAINS\backend\access\common\indextemplate.c Num_CIBs="4" Name="backend.access.common.indextemplate.c" Id="2"/>
  </Component>
  <Component Arch_XML_file="comp-3-backend.access.common.printtup.c.xml">
    <Program_File>$MAINS\backend\access\common\printtup.c Num_CIBs="13" Name="backend.access.common.printtup.c" Id="3"/>
  </Component>
  <Component Arch_XML_file="comp-4-backend.access.common.relations.c.xml">
    <Program_File>$MAINS\backend\access\common\relations.c Num_CIBs="17" Name="backend.access.common.relations.c" Id="4"/>
  </Component>
  <Component Arch_XML_file="comp-5-backend.access.common.scankey.c.xml">
    <Program_File>$MAINS\backend\access\common\scankey.c Num_CIBs="4" Name="backend.access.common.scankey.c" Id="5"/>
  </Component>
  <Component Arch_XML_file="comp-6-backend.access.common.tupdesc.c.xml">
    <Program_File>$MAINS\backend\access\common\tupdesc.c Num_CIBs="12" Name="backend.access.common.tupdesc.c" Id="6"/>
  </Component>
  <Component Arch_XML_file="comp-7-backend.access.gin.ginarrayproc.proc.c.xml">
    <Program_File>$MAINS\backend\access\gin\ginarrayproc.proc.c Num_CIBs="4" Name="backend.access.gin.ginarrayproc.proc.c" Id="7"/>
  </Component>
  <Component Arch_XML_file="comp-8-backend.access.gin.ginbtree.c.xml">
    <Program_File>$MAINS\backend\access\gin\ginbtree.c Num_CIBs="7" Name="backend.access.gin.ginbtree.c" Id="8"/>
  </Component>
  <Component Arch_XML_file="comp-9-backend.access.gin.ginbulk.c.xml">
    <Program_File>$MAINS\backend\access\gin\ginbulk.c Num_CIBs="11" Name="backend.access.gin.ginbulk.c" Id="9"/>
  </Component>
  <Component Arch_XML_file="comp-10-backend.access.gin.gindatapage.c.xml">
    <Program_File>$MAINS\backend\access\gin\gindatapage.c Num_CIBs="18" Name="backend.access.gin.gindatapage.c" Id="10"/>
  </Component>
</Components>
```

Figure A.3: PostgreSQL version 8.4.4 system structure.
Similarly, Figure A.4 presents a component structure XML document. The document represents the Component `pg_dump` located in `/backend/bin/pg_dump/` folder of the PostgreSQL database project.

![XML representation of `pg_dump` component structure](image)

Figure A.4: The structure of a sample component (`pg_dump`) of the PostgreSQL project.
A.2 Control Flow Monitor

Here, we present some generated and utilized artifacts by our control flow monitor.

A.2.1 Signature Injection

Figure A.5 displays a sample file of the PostgreSQL database software project. The file is `freespace.c` located in the folder `/backend/storage/freespace/`. The file is automatically injected by control flow signatures. The procedure `dOSpy()` exports the runtime signatures in the file.

```c
static int
fsm_set_avail(int page, int slot, int value)
{
    int newSlot = -1;

dOSpy("340,18,0");
buf = fsm_readbuf(rel, addr, true);
dOSpy("340,18,1");
LockBuffer(buf, BUFFER_LOCK_EXCLUSIVE);

dOSpy("340,18,2");
page = BufferGetPage(buf);

dOSpy("340,18,3");
if (fsm_set_avail(page, slot, value))
    dOSpy("340,18,4");
MarkBufferDirty(buf);

if (minValue != 0)
{
    /* Search while we still hold the lock */
dOSpy("340,18,5");
    newSlot = fsm_search_avail(buf, minValue,
                               addr.level == FSM_BOTTOM_LEVEL,
                               true);
}

dOSpy("340,18,6");
UnlockReleaseBuffer(buf);
return newSlot;
```

Figure A.5: Automatically injected signatures in the PostgreSQL database software.
Figure A.6 shows the implementation of the procedure \textit{dOSpy()}. This procedure exports the runtime signatures to the output stream and manages the output data based on

Figure A.6: The procedure \textit{dOSpy()} that exports the runtime signatures to the output stream.
the operating system forking. Managing the output data based on the operating system forking is essential to control the log counters of each concurrent process and maintain appropriate sequences of log records. The procedure $dOSpy()$ is called at each injected line of code by our control flow monitor. The call parameter at each line is the control flow signature of the corresponding connector ID.

A.2.2 Automatic Execution Toolkit

This subsection presents some samples of the shell scripts and SQL files used for automatic execution of the PostgreSQL database versions and experimental data sets.

Figure A.7 displays part of a shell script file for automatically compiling and operating

```
for i in 1 2 3 4 5 6 7 8
do
    postgres -D $HOME/pqsql/data 2>> $HOME/pgexp/ateflog$i &
    tail -1 $HOME/pgexp/ateflog$i
    if [ $j -gt 0 ]
    then
        find ExecBsh/ -name "*.bsh" | wc -l
        rm -f $HOME/pgexp/ExecBsh/v$i/*.bsh &> $HOME/pgexp/ateferr
    fi

    case "$j" in
        '1') #-------- RUN 1
            createdb -T template0 atef_temp1 >> $HOME/pgexp/ateflog$i
            createdb atef$i1 >> $HOME/pgexp/ateflog$i
            dropdb atef_temp1 >> $HOME/pgexp/ateflog$i
            dropdb atef$i1 >> $HOME/pgexp/ateflog$i
        ;;
        '2') #-------- RUN 2
            psql -f $HOME/pgexp/pg2.sql atefdb$i1 >> $HOME/pgexp/ateflog$i
            tail -1 $HOME/pgexp/ateflog$i
        ;;
        '3') #-------- RUN 3
            psql -f $HOME/pgexp/pg3.sql atefds$i1 >> $HOME/pgexp/ateflog$i
            tail -2 $HOME/pgexp/ateflog$i
        ;;
        '4') #-------- RUN 4
            psql -f $HOME/pgexp/pg4.sql atefds$i1 >> $HOME/pgexp/ateflog$i
        ;;
        '5') #-------- RUN 5
            psql -f $HOME/pgexp/pg5.sql atefds$i1 >> $HOME/pgexp/ateflog$i
            tail -2 $HOME/pgexp/ateflog$i
    esac
```

Figure A.7: Part of the shell script of the automatic operation toolkit.
the different PostgreSQL database versions and managing their outputs in a folder structure. This script is called through other shell programs that perform higher level tasks of the automatic execution including folder cleaning, switching between experimental data sets, and some other tasks.

Figure A.8 presents a sample SQL file used as experimental data set during the automatic execution of the PostgreSQL database server versions. The SQL file performs some database server commands, creates tables, inserts data into them, and executes queries on these data.

```
1 \l \d D \dT \dp \copyright
2 CREATE TABLE cities (
3   name varchar(80),
4   province varchar(80)
5 )
6 CREATE TABLE weather {
7   city varchar(80),
8   temp_lo int, -- low temperature
9   temp_hi int, -- high temperature
10   pcp real, -- precipitation
11   date date
12 }
13 INSERT INTO cities VALUES ('Toronto', 'Ontario');
14 INSERT INTO cities VALUES ('Montreal', 'Quebec');
15 INSERT INTO cities VALUES ('Vancouver', 'British Columbia');
16 INSERT INTO weather VALUES ('Toronto', 33, 50, 0.25, '1994-11-27');
17 INSERT INTO weather VALUES ('Montreal', 25, 50, 0.25, '1994-11-27');
18 INSERT INTO weather VALUES ('Vancouver', 56, 80, 0.25, '1994-11-27');
\c cities
\c weather
```

Figure A.8: A sample SQL file that is passed as a parameter to the automatic execution toolkit.
A.2.3 Exported Runtime Signatures

Here, we show some exported runtime signatures by the executed software versions. These runtime signatures are used by the control flow monitor to validate the control flow transitions and export runtime error log.

Figure A.9 shows some exported runtime signatures by the executed software versions. The signatures are organized in files that are named after the concurrent process Id and file counters. In each file, a list of runtime signatures represent the control flow path during an execution interval of this process. Each line in this file indicates a control flow transition.

![Figure A.9: The exported runtime signatures by an executed version of the PostgreSQL database.](image)
A.3 Failure Mode Prediction

We present the actual execution times of the system runs with respect to the experimental evaluation of our failure mode prediction technique in Table A.1.
### A.3. FAILURE MODE PREDICTION

<table>
<thead>
<tr>
<th></th>
<th>run 0</th>
<th>run 1</th>
<th>run 2</th>
<th>run 3</th>
<th>run 4</th>
<th>run 5</th>
<th>run 6</th>
<th>run 7</th>
<th>run 8</th>
<th>run 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_0$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$E_1$</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>25</td>
<td>8</td>
<td>12</td>
<td>23</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$E_2$</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>18</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$E_3$</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$E_4$</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>20</td>
<td>23</td>
<td>7</td>
<td>3</td>
<td>17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$E_5$</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>991</td>
<td>2</td>
</tr>
<tr>
<td>$E_6$</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$E_7$</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$E_8$</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>$E_9$</td>
<td>1463</td>
<td>1250</td>
<td>1275</td>
<td>1341</td>
<td>1327</td>
<td>717</td>
<td>721</td>
<td>3</td>
<td>22749</td>
<td>14399</td>
</tr>
</tbody>
</table>

Table A.1: The actual execution times of the systems runs against the experimental data sets.

The execution time of the system runs (columns) against the experimental data sets (rows) are approximated and displayed in seconds. These execution times are used to capture the late and early service delivery failures described in Chapter 5.