2004

Empirical assessment of architecture-based reliability of open-source software

Ranganath Perugupalli
West Virginia University

Empirical Assessment of Architecture-Based Reliability of Open-Source Software

Ranganath Perugupalli

Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
In partial fulfillment of the requirements
For the degree of

Master of Science
In
Computer Science

Katerina Goseva-Popstojanova, Ph.D., Chair
James D. Mooney, Ph.D.
Hany H. Ammar, Ph.D.

Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2004

ABSTRACT

Empirical Assessment of Architecture-Based Reliability of Open-Source Software

Ranganath Perugupalli

A number of analytical models have been proposed earlier for quantifying software reliability. Some of these models estimate the failure behavior of the software using *black-box* testing, which treats the software as a monolithic whole. With the evolution of component based software development, the necessity to use white-box testing increased. A few architecture-based reliability models, which use white-box approach, were proposed earlier and they have been validated using several small case studies and proved to be correct. However, there is a dearth of large-scale empirical data used for reliability analysis. This thesis enriches the empirical knowledge in software reliability engineering. We use a real, large-scale case study, GCC compiler, for our experiments. To the best of out knowledge, this is the most comprehensive case study ever used for software reliability analysis. The software is instrumented with a profiler, to extract the execution profiles of the test cases. The execution profiles form the basis for building the operational profile of the system, which describes the software usage. The test case failures are traced back to the faults in the source code to analyze the failure behavior of the components. These results are used to estimate the reliability of the software, as well as the uncertainty in the reliability analysis using entropy.

Keywords – software architecture, software reliability, open-source software, operational profile, failure analysis, entropy, uncertainty, cvs, change logs, profiler, bugzilla.

DEDICATION

I am honored to dedicate this publication to my beloved father, Shri Anantha Rama Sarma, who did more than he could for me. I express my deep love to my Uncle, late Shri Subrahmanya Sastri, who encouraged me to pursue higher studies. I wish that his soul rests in peace after his long pain and suffering. I express my deep love and affection to my mother Padmavathi and brothers Ramakrishna and Ravi, and my aunt Mythili for their continuous support and motivation.

ACKNOWLEDGEMENTS

I express my sincere thanks to Dr.Katerina Goseva Popstojanova, for giving me the opportunity to work with her. I thank her for her motivation and guidance during my research. I would like to thank my committee members Dr. James D. Mooney, and Dr. Hany H. Ammar, for their valuable time, contribution and support in this course of study. I thank my research colleague, Sunil for his help during my research work. I express my special thanks to Maggie* Hamill, for her support and contribution in my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A number of analytical models have been proposed earlier for quantifying software reliability. Some of these models talked about the reliability growth at the testing phase [38]. The software reliability is estimated using *black-box* testing with a randomly chosen set of test cases. The *black-box* models treat the software as a monolithic whole. These models care only about the outcome of the testing and do not consider the internal structure of the software. With the evolution of component-based software development software-reuse is of utmost importance to the modern day developers. The black-box approach was proved to be inappropriate for this kind of systems. We need to employ white-box model for these component-based systems, which also consider the information about the architecture of the software at the component level. The methodology to architecture-based reliability assessment proposed in [20] is described here. Figure 1.1 depicts the graphical representation of the methodology. In order to estimate the software reliability using the architecture-based model we need to know

- The software architecture described by the flow of control among components in the system
- The software usage described by its component interactions determined by the operational profile.
- The software failure behavior described by reliabilities of the components

In this chapter we explain the architecture-based approach to find the software architecture, software usage and the software failure behavior.

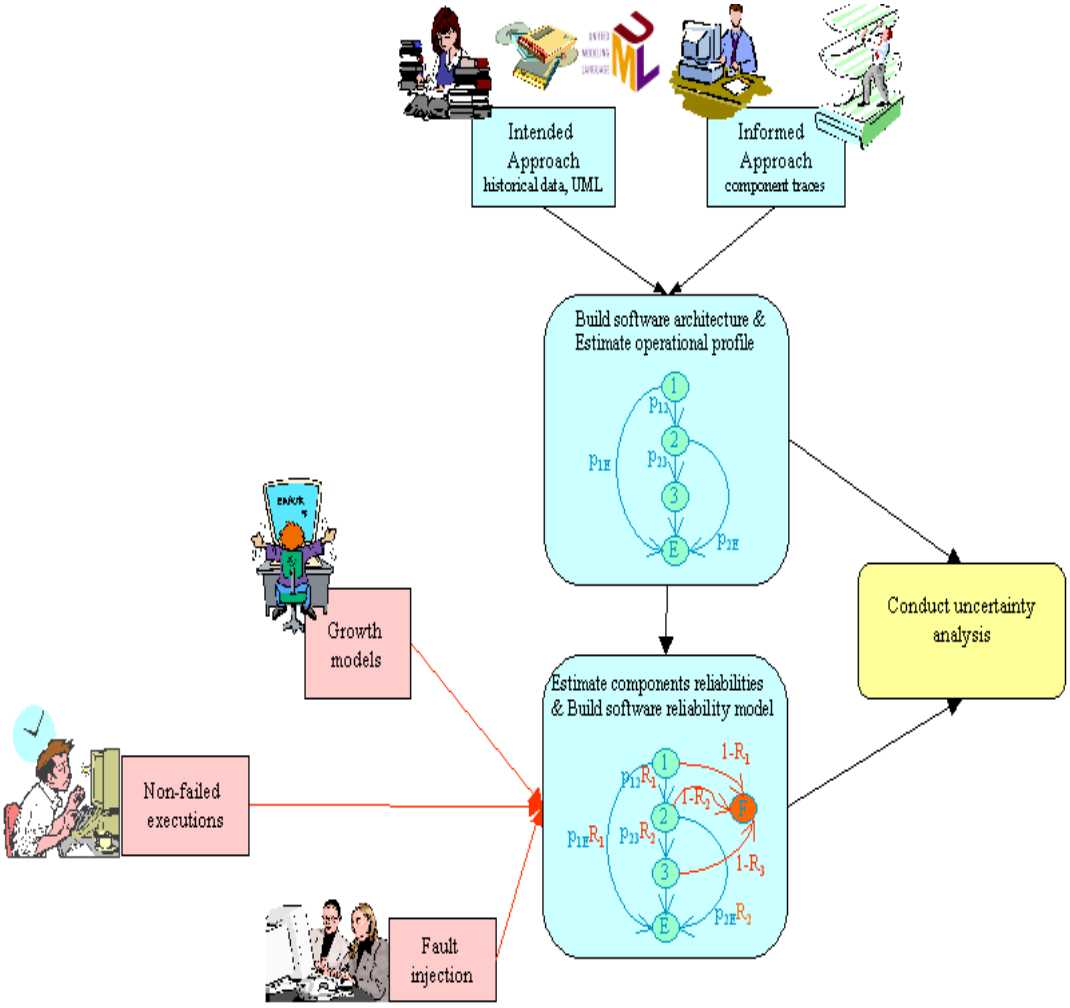


Figure 1.1: Architecture-Based Methodology for Reliability Analysis

A state-based approach is used to build the architecture-based reliability model [7] and [8]. The architecture-based reliability model is appropriate for large component-based software. The architecture of software is defined by the way the components in the system interact with each other. The model uses the control flow graph as a representation of the software architecture. The states in the diagram represent the components in the system and the arcs represent the interaction between the components in the form of control transfer. It is assumed that the component interactions have the Markov property. The software architecture is modeled with discrete time Markov chain (DTMC). $P = [p_{ij}]$ is the transition probability matrix of the Markov chain, where p_{ij} is the probability that control is transferred from *i* to *j*. The Markov chain is constructed in two

phases. The *structural phase* establishes the software architecture, using different abstraction levels with the data obtained from the requirement-specification or the static metrics obtained from the lexically based code parsers. This phase does not consider the component interactions during the execution. The *dynamic statistical phase* estimates the transition probabilities of the components. The component interactions depend on the operational profile of the system. Depending on the phase of development of the software, the dynamic behavior of the software can be found using either the Unified Modelling Language (UML, in early stages of development) or from the test coverage tools (in integration phase). There are two different approaches that are generally followed for building the Markov chain model.

- ***Intended Approach*** is used if the software is in its early phase of development. The software architecture is estimated using the information obtained from the design and specification documents or using some historical data from similar products. The object-oriented systems use UML as a standard design tool. We can use *use-case* diagrams and *sequence* diagrams obtained from the UML specifications to make an estimation of software architecture [39]. The *sequence* diagrams depict the interaction between the components (mentioned in *use-case* diagram) in the form of message passing. The transition probability of the component i to component j is given by $p_{ij} = (n_{ij} / n_i)$, where n_{ij} is the number of times component i sends messages to j and n_i is the total number of messages from component i.

- ***Informed Approach*** is used if the software is in later phases of development, in which the source code is available and accessible. The dynamic behavior of the components is estimated using the testing tools and the source code. Profiling tools [4] and test coverage tools [40] are used to obtain the component traces during the test case executions. The transition probabilities are obtained using the frequency counts of the component interactions.

K. Goseva Popstojanova and Sunil. K. Kamavaram applied this methodology to find the uncertainty in reliability estimation using the European Space Agency (ESA) software [20]. The ESA software, which has 10,000 lines of code, is also small compared to some industrial software applications. However, in spite of its importance, there have been very few efforts on applying large-scale industry level empirical case studies in the filed of reliability. Although researchers like David Leon, Andy Podgurski used large-scale software systems in [12], [27], [28] and [30], these studies are focused on execution profiles rather than the reliability of the software. The reason for not having many of such contributions is that locating and gaining access to the large-scale software is difficult and the process of collecting and analyzing the necessary data is very time consuming and also very expensive.

The main motivation for this thesis is the dearth of empirical data available on large-scale software systems in the field of software reliability. This thesis is focused on using large-scale case studies to validate the architecture-based reliability models, as well as on contributing towards the usage of larger case studies the field of software reliability. We use GCC, a GNU open source compiler, which is being used for several years and has more than 30 versions released over a period of 7 years. GCC has more than 800,000 lines of source code written in C and is the most comprehensive case study ever used for the reliability analysis.

1.1 Related Work

This thesis emphasizes the usage of large-scale empirical case studies for software reliability analysis. We implement the architecture-based methodology for uncertainty analysis of software reliability proposed in [7] and [20] to estimate the reliability of the system and to study the uncertainty analysis of reliability using entropy. We implement the white-box approach to estimate the operational profile of the software. This approach is different from the black-box approach for software reliability modeling, where the system is considered as a monolithic entity [23]. In black-box approach only the interaction of the system with the out side world are considered. We use the executing

profiles generated during testing to analyze the failure behavior of the system (see Chapter 6).

The difficulties in handling the large-scale empirical case studies were discussed by Thomas J.Ostrand and Elaine J. Weyuker in [24]. We are using GCC compiler, which is a GNU open source project. It has 300 source files and 800,000 lines of C code, which is much bigger than the case study they used (an inventory control system with 500,000 lines of code) in [24]. The version management systems maintained by the developers for these projects are huge and difficult to extract and analyze [24]. GCC maintains a CVS repository in the form *Change-Log* files, which contain the changes made to the source over a period of time. The problem with these *Change-Log*'s and the MR (modification requests) data repositories that were mentioned in [24] is that they are not intended for the purpose of the fault detection, so it is very difficult for us to find the information we need. There can be different kinds of changes in these log files such as fixing faults, code enhancements, code modifications, new code, and also documentation change. It is difficult to find out which of those changes are initiated because of a fault. In [24] Thomas J.Ostrand and Elaine J. Weyuker made an assumption that, if just one or two files were changed then it was likely a fault, while if more than two files were changed then it is more likely a code modification or an enhancement. Instead of making this kind of assumption, we propose more accurate methods to identify faults (see Chapter 7). In this thesis, we were successful in extracting the fault information from the *Change-Log*'s and finding the critical components in the system that failed most number of times. This information is useful in making decision on allocating the testing efforts for the components in the system.

Andy Podgurski, Jiayang Sun and Bin Wang used GCC in [12] to come up with an automated support for classifying reported software failures in order to facilitate prioritization and diagnosing the faults. The main intention of their research was to provide the developer with the classification of failures and failure clusters, so that the developer can plan his testing efforts accordingly. Andy Podgurski and David Leon used GCC 2.95.2 and the regression test suite provided with GCC 3.0.2 to conduct their

experiments. C proper part of GCC was used for the experiments. The open source projects like GCC do not have the sophisticated bug-reporting system like some of the commercial software projects. Andy Podgurski and David Leon used *Change-Logs* provided with GCC to map failures to faults and could manage to map most of the failures to corresponding faults using these log files. Andy Podgurski and David Leon implemented the method "*Execute tests on newer versions & search logs*" (explained in chapter 6), to classify the remaining failures. We use GCC 3.2.3 and the test suite of GCC 3.3.3 for our research. We use execution profiles of the test cases and the *Change-Log*s of GCC to come up with the reliability estimates for the components that tells us which components have higher reliability and which of them are more fault prone and need more testing efforts. In [30], David Leon, Andy Podgurski used large-scale open source projects like GCC*, Jikes* and *javac* to compare four different techniques for test case filtering: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling. David Leon, Andy Podgurski used the regression test suite provided with GCC 3.0.2, which contains test cases for defects present in GCC 2.95.2. 136 test cases were failed out of 3,333 test cases they have executed. Executions were profiled using *gcov*, a basic block profiler provided with GCC. Using these execution profiles, David Leon, Andy Podgurski manually classified the failures into groups of failures that were assumed to have same cause. There failures were mapped to 27 defects in the system.

Swapna S.Gokhale, W. Eric Wong and S.Trivedi conducted experiments on large-scale empirical case study in [23] to come up with an analytical approach to architecture-based reliability prediction. The reliability model was represented as a discrete time Markov chain (DTMC). Execution profiles generated during extensive testing were used to find the branch probabilities of the DTMC. All the experiments were conducted on an application called as SHARPE (Symbolic Hierarchical Automated Reliability Predictor), which is used to solve stochastic models of reliability, performance and performability [23]. It has multiple releases and change information associated with each release. SHARPE has a total of 373 functions and 35,412 lines of C code, which is very small compared to GCC that has more than 800,000 lines of code. Swapna S.Gokhale, W. Eric

Wong and S.Trivedi found the failure behavior of the component by a time-dependent failure intensity, which can be determined using test coverage and fault density approach [25]. A dataflow-coverage testing tool called ATAC (Automatic Test Analyzer in C) was used to find the test coverage, on 735 test cases that were created by the developers to test SHARPE. ATAC not only runs the test cases but also generates execution profiles. An assumption that when a function X calls another function Y control is eventually transferred back to function X, was used for the experiments.

Thomas J.Ostrand and Elaine J. Weyuker conducted experiments on a large-scale fault-reporting database that is collected for all production systems at AT&T [33]. This was an inventory tracking system that has 13 releases produced over a period of several years. The current version has 1,974 files written in JAVA, with a total of 500,000 lines of code. Whenever a fault is identified in the system an entry is made in the database associated with the corresponding software. The entry includes, the stage of the development that the problem was identified, the release version of the program and the severity of the problem. The data is similar to the modification request (MR) data used in [24]. This data was used to come up with the fault distribution among the different files in the system. In addition to the fault distributions among the files, Thomas J.Ostrand, Elaine J. Weyuker addressed many issues in [33] like, affect of the module size on the fault density, persistence of failures between different releases of the software and whether newly written files were more fault-prone then the old files written for the earlier version. A *module* is the basic code component of the system. The goal of the experiment was to identify the files that were more fault-prone and could be used as predictors of fault-proneness of the system. Even though it was commercial software, which is considered to be highly reliable, Thomas J.Ostrand and Elaine J. Weyuker mentioned that finding faults from the database was hard because the data was not well organized. Our research does differ from what Thomas J.Ostrand and Elaine J. Weyuker did in the sense that we used dynamic data from testing to come up with the failures and them mapped them to the faults using the *Change-Logs* (see chapter 6) to estimate the fault-proneness of the components in the system. With the experiments conducted on the AT&T software, Thomas J. Ostrand, Elaine J. Weyuker found that for each release, the “faults were

always heavily concentrated in a relatively small number of files". Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell conducted some more experiments in [32] on the same case study (AT&T inventory tracking system) to find out which files in the software system are most likely to contain the largest number of faults. The AT&T software has a version tracking system maintained through out the life cycle of the project. The system contains the MR (modification request) entries, which has the changes made to the different files in the system. Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell used static analysis of the version tracking system to find the fault-prone files. Finding the faults using the MR entries was not straight forward because an MR may contain a change that was initiated because of a fault, an enhancement or change in the specifications, and it was difficult to differentiate between different kinds of changes. An assumption that "if only one or two files were changed by the modification request (MR), then it was likely a fault, while if more than two files were affected, it was likely not a fault", was made through out the experiments. Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell concluded that 20% of the total files, in which they found most critical faults, were constituted to 83% of the total system size.

Norman E. Fenton and Niclas Ohlsson in [31] discussed the dearth of empirical data in the field of software engineering. Norman E. Fenton and Niclas Ohlsson, in their experiments found that a very small number of modules in the system contain most of faults discovered in the testing phase as well as the normal operations. However, contradicting the conclusions made by Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell in [32], [33], Norman E. Fenton and Niclas Ohlsson found that the fault-proneness of the modules do not depend on their size or complexity. This finding proved that, the most widely used fault density measures and the metric studies based on those measures are flawed. Norman E. Fenton and Niclas Ohlsson also stated in [31] that, those modules that are most fault-prone pre-release are among the ones that are least fault-prone post-release. All these observations were based on the experiments conducted on the empirical data obtained from a large telecommunication application from Ericsson Telecom AB. Michael R. Lyu, Zubin Huang, Sam K. S. Sze and Xia Cai discussed the problem of limited empirical data available in the literature, to evaluate the effectiveness

of software testing and fault tolerance in [29]. Mutation testing [29] was used to evaluate the effectiveness of software testing and software fault tolerance. Mutants were created by injecting faults in to the software. Michael R. Lyu, Zubin Huang, Sam K. S. Sze and Xia Cai stated that coverage testing [29] is an effective way of fault detection. However it was also mentioned that testing coverage is not as effective as mutant coverage to evaluate the testing quality. An industry-scale critical flight control system was developed to conduct the experiments.

1.2 Problem Statement

This thesis focuses on empirical assessment of architecture-based methodology for software reliability analysis, using large real life empirical case studies. It addresses two critical problems associated with handling large-scale empirical case studies for architecture-based reliability assessment. The first problem is to develop an efficient way for building the operational profile of the software from large number of huge execution profiles obtained during testing. The second problem is to automate the analysis of the failure behavior of components (i.e., to identify faults that led to failures), using the software artifacts such as change logs and CVS logs, which are not specifically made for the purpose of failure analysis.

1.3 Contributions

The most important thing that differentiates our research from most of the other work presented in related work is the size of the case study we are using for the reliability analysis. Despite the importance of using large-scale industrial software's for the reliability analysis, there have been very limited efforts in this area. To the best of our knowledge, GCC 3.2.3, a GNU open source compiler, with approximately 800,000 lines of C source code, is the largest case study ever used for the study of software reliability analysis. Using such a large case study for reliability analysis, itself is a major

contribution, considering the limited empirical data available in the area. The contributions of this thesis are summarized as follows.

- The main contribution of this thesis to the architecture based approach for reliability assessment proposed in [7] and [8] is to validate the methodology by implementing it on large-scale object based case study. The architecture-based model given in [20] and [8] for the software reliability assessment considers the usage of the software described by its operational profile. Building the operational profile for such a large case study was not trivial. We used test cases provided with GCC to generate execution profiles, which are used to estimate the dynamic behavior of the software. The profiler gives the execution details at function level. Mapping these functions to components was a hard task, because the documentation available was old and not sufficient to do the mapping. We were able to build the operational profile for GCC at the component level and find the transition probabilities of the components. Unlike the manual process followed by David Leon, Andy Podgurski in [30] to analyze the execution profiles, we automated the process of parsing the execution profiles and storing the data in relational database, which made the calculation of transition probabilities efficient, even though the profiles we have 2126 execution profiles, each with more than 2500 function calls.

- Another major contribution of the thesis is in finding the faults in the system that caused the failures in test case executions. After identifying the failed test cases we used the *Test case Change-Logs* and *Source code Change-Logs* provided with the GCC source code. The problem with these *Change-Log* files is similar to what Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell had with MR's (modification requests) in [32]. The *Change-Log* files were not created for the purpose of finding the faults in the system. We employed three methods to map the failures to faults in the source code; Searching *test case Change-Logs* and *GCC Change-Logs*, Search the bug-tracking database Bugzilla, Execute tests on newer versions & search logs. The last method (Execute tests on newer versions &

search logs) was first used by Andy Podgurski and David Leon in [12]. The *Test-case Change-Log* has approximately 3500 entries and the *Source-code Change-Logs* have approximately 19,300 entries. Searching through these files manually to map failures to faults is time consuming and error prone. We have automated the process of searching through both change logs to map the failures to the corresponding faults. Since most of the open source software maintains the same format for the *Change-Log* files the automation saves a lot of time and effort.

The failures were mapped to faults in GCC. The mapping was done at both file level and component level. We observed that there are very few files where most of the faults are concentrated. This observation strengthens the argument made by Thomas J.Ostrand and Elaine J. Weyuker in [32] and [33] that "the faults were always heavily concentrated in a relatively small number of files". The overall system reliability was calculated using the method proposed by K. Goseva Popstojanova and K. S. Trivedi in [7]. We compare the value of the reliability with the value we got from the *black-box* method and found that the value is relatively accurate. We have conducted uncertainty analysis using entropy, on GCC, which was proposed by S. K. Kamavaram and K. Goseva Popstojanova in [9]. We observed that some of the components are more uncertain, and thus more critical than the rest of the components.

1.4 Thesis Outline

This thesis presents an empirical analysis of architecture-based analysis of software reliability. The current chapter gives a brief introduction to the architecture-based methodology for reliability analysis in component-based systems, describes the related work done in this area, and explains the contribution. Chapter 2 presents the approach we followed to implement the architecture-based methodology. Chapter 3 describes the case study we are using. Chapter 4 gives a detailed explanation about the experimental setup. Chapter 5 describes about finding the operational profile. Chapter 6 explains the mapping of failures to faults. Chapter 7 presents uncertainty analysis based on entropy. Finally chapter 8 presents the conclusions.

Chapter 2

Our Approach

We used GCC, a GNU open source compiler for our experiments. Open source projects are suitable for our experiments on reliability analysis using white-box approach because many software artifacts are available, like

- Source code
- Requirements and design documentation
- Test suites, with an oracle
- CVS logs which contain change information between different version releases
- GCC *Change-Logs* and test case *Change-Logs*

Figure 2.1 describes our empirical approach for the architecture based reliability analysis. The Figure 2.1 explains the procedure we followed to extract the information we need for the reliability analysis like software architecture, software usage and software failure behavior.

Since we are using white-box approach for the reliability analysis, we need to know what part of the code has been executed, which functions are called, which functions take much time to execute. For this purpose we need a profiler that tracks the executions of the software and gives us the data we need. We used *gprof*, a GNU open source profiler that is used specially to profile applications written in C and C++. GCC is instrumented wit *gprof*. We chose to use the test suite provided with GCC to get the execution profile. We run the test cases with *Dejagnu,* a GNU testing tool. The generated execution profiles are stored in an ORACLE database so that they can be used easily and efficiently.

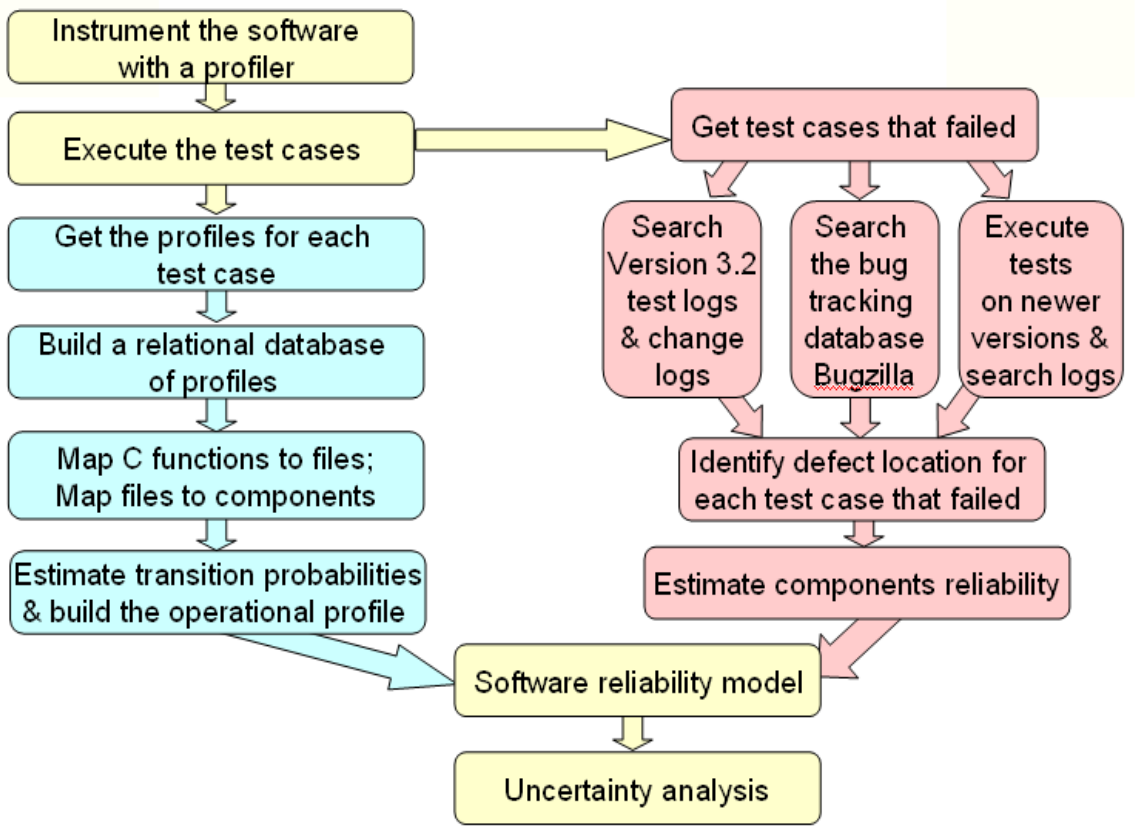


Figure 2.1: Our Approach

2.1 Software Architecture and Usage

The software architecture and the usage are reflected by the operational profile of the software. We used the execution profiles we got from the test executions as the basis for building GCC architecture. The information in the profiles also gives us the usage of different components of GCC. The profiler gives us the execution profile at the function level. We have 1759 unique functions that were invoked during the execution of all the 2126 test cases. It is difficult to build the software architectural model using so many states. We mapped these functions to files in the source code so that the number is more manageable. We have 108 files that these functions belong to. Building the architectural model using 108 states is also not trivial. We wanted to reduce this number further by

mapping these files to components. Component is a part of compiler that is dedicated to a particular functionality. We have 17 components in our system. We mapped 108 files we have to these 17 components to build the architecture of GCC. We calculated the values for the transition probabilities using the values from the database. The database is built in such a way that it has information about the executions at function level, file level and also at the component levels. We built the operational profile of GCC based on the test case execution profiles.

2.2 Software Failure Behavior

Even though the test suite does not reflect the system's usage perfectly, this test suite has wide variety of test cases that tests different features of gcc like, variables, language dependent structures, and memory allocations. The most important reason for choosing the test suite is that they have the failure information associated with the test case failures, which will be used as a oracle to find the failure behavior of the system. We employed three methods to find the defect information about the failures.

- *Searching test logs & change logs***:** We searched the GCC *Change-Log* files and the test case *Change-Log* files to find the location of the failure. The details of implementing this method and sample log files are given in Section 6.1. The whole process is automated using *awk*. We could find 41 defects with this method. This is the most efficient method among the three.

- *Search the bug-tracking database Bugzilla*: Bugzilla is a "Defect Tracking System" or "Bug-Tracking System" [13]. Bugzilla has a big database which has information about the failures in the system. The failures may be due to the failures of test case or operational failures that users find after deployment. The test case *Change-Log* files has a PR (problem report) number associated with each test case. Bugzilla used this PR number to index the failures related to the test case executions.We searched the bugzilla databe with the PR numbers we have for the failed test cases. We could find only 3 defects with this method. The

details of the implementation and a sample bugzilla database are given in Section 6.2.

- *Execute tests on newer versions & search logs*: The failed test cases are tested against the newer versions so that we could find out when the bug was fixed. After finding the version in which the bug was fixed, we repeat the first method to trace the location of the defect. This method is more efficient than the second one. We could find 20 defects with this method.

2.3 Calculating compontnt and system reliabilities

"The reliability of component $\boldsymbol{i}$ is the probability $\boldsymbol{R_i}$ that the component performs its function correctly" [20]. We already have the information about the non-failed test cases and the failure behavior of the system was determined in section 2.2.2. We found the mean value of the component reliabilities using the failure information of the test cases and the number of non-failed test cases. Since the number of failures for each component is very small compared to the number of executions for each component, the values for the component reliabilities are very high. The system reliability is calculated using the method explained in [20]. The reliability is also calculated using the black box approach by dividing the number of failed test cases over the total number of test cases. These two values are compared and the error is estimated.

2.4 Uncertainty analysis

There will be a certain amount of uncertainty in the reliability calculation even though the mathematical model is accurate [7]. The uncertainty of the operational profile and the reliability model are analyzed using the concept of source entropy. Entropy is a very well known concept in *information theory*. It cannot estimate the reliability value. We used entropy to calculate the amount of uncertainty in GCC, which is represented as a Markov chain. The range of the value is $0 \leq H(S) \leq log\ (n)$ [9]**.** We used conditional entropy to calculate the component uncertainties. We also found the expected execution rates of all components.

Chapter 3

Description of case study

3.1 Introduction to GCC

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for several major programming languages including C, C++, Objective-C, Java, Fortran, and Ada [1]. The part of compiler that is specific to a particular language is called the "front-end". GCC also supports front-ends for Pascal, Mercury and Cobol in addition the above mentioned languages. Initially GCC was refered to as 'GNU C compiler' when it was used only to compile C programs. Even now we use the same definition when we refer to the compilation of C programs or when we speak of the *language-independent* component of GCC, which is the code that is used commonly for all the languages that it supports. The majority of the compiler optimizers are included in the language independent component of GCC. It also includes all the 'back-ends', which are used to generate machine code for various processors.

GCC is an open source software that is available for free. The different versions released by the GCC community as of August 2004 can be seen in Table 3.1. Source code for each version is available to download from different mirror sites of GCC. GNU also maintains a CVS (Concurrent Version System) repository to avail the users to download the source code.

Release	Release date
GCC 3.4.1	July 1, 2004
GCC 3.4.0	April 18, 2004
GCC 3.3.4	May 31, 2004
GCC 3.3.3	February 14, 2004
GCC 3.3.2	October 17, 2003
GCC 3.3.1	August 8, 2003
GCC 3.3	May 13, 2003
GCC 3.2.3	April 22, 2003
GCC 3.2.2	February 05, 2003
GCC 3.2.1	November 19, 2002
GCC 3.2	August 14, 2002
GCC 3.1.1	July 25, 2002
GCC 3.1	May 15, 2002
GCC 3.0.4	February 20, 2002
GCC 3.0.3	December 20, 2001
GCC 3.0.2	October 25, 2001
GCC 3.0.1	August 20, 2001
GCC 3.0	June 18, 2001
GCC 2.95.3	March 16, 2001
GCC 2.95.2	October 24, 1999
GCC 2.95.1	August 19, 1999
GCC 2.95	July 31, 1999

Table 3.1: Versions of GCC released as of August 2004

3.2 Size of GCC

To the best of our knowledge, GCC is the biggest case study ever used for empirical software reliability estimation. In our experiments we used the *C proper* part (the part of GCC that compiles programs written only in C) of GCC. The C proper part itself has 300 source files written in 12 different languages and has approximately 800,000 lines of ANSI C code [2]. These files include both the programming and scripting languages. Table 3.2 contains the list of languages and lines of code written in each of those languages for the version GCC-2.96-20000731. This table shows how large the case study actually is.

LANGUAGE	***LOC***
ANSI C	789,901
CPP	126,738
YACC	19,272
SH	17,993
ASM	14,559
LISP	7,161
FORTRAN	3,814
EXPECT	3,705
SED	310
PERL	144
OBJC	479
Total	984,076

Table 3.2: Details of the source code for GCC-2.96-20000731

3.3 Process of compilation

The whole process of compilation in GCC is controlled by a single file named toplev.c. The process of compilation is implemented in multiple passes [3]. In addition to sequencing all the passes this file has many additonal responsibilites such as initialization, decoding arguments, opening and closing files. The parsing pass is called first from the toplev.c. The parsing pass parses the file and generates the high level tree representation. The tree representation is converted into RTL (Register Transfer Language) intermediate code using the files *expr.c, expmed.c and stmt.c*. After finishing the parsing of the function-definition the parsing pass calls the function `rest_of_compilation` in *toplev.c*. The function `rest_of_compilation` is responsible for finishing the rest of the compilation process and printing the assembly code for that function definition. The parsing pass calls the function `rest_of_decl_compilation` when it reads a top-level declaration. All the other passes are called by `rest_of_compilation` in sequence. Once the function definition is compiled the storage used for compilation is freed except for the inline functions. The process of compilation is performed in 20 different passes including the parsing pass.

3.4 Test cases

GCC has a regression test suite maintained to ensure the quality of the software over a period of time. This test suite comes with the full distribution of GCC. New test cases are added to the regression test suite with each release of GCC. When a new version of GCC is released, normal users as well as the developers test it against different programs. Some of the programs may give warnings or fail to give the expected results. When an unexpected output is found, the user tries to locate the bug and fixes it. All such test cases are added to the regression test suite. The new test cases that were added would be available with the next released version. In this way the developers make sure that the bug will not be present in the newer version. Since we must determine the locations of the bugs, it is best to use test cases from the latest version and test them against the old version. Most of the test cases that were added to the new version will fail on the older

version. For example we conducted all of our experiments on GCC 3.2.3, but we used test cases from GCC 3.3.3, which includes test cases from four newer versions. Figure 3.1 depicts the process involved.

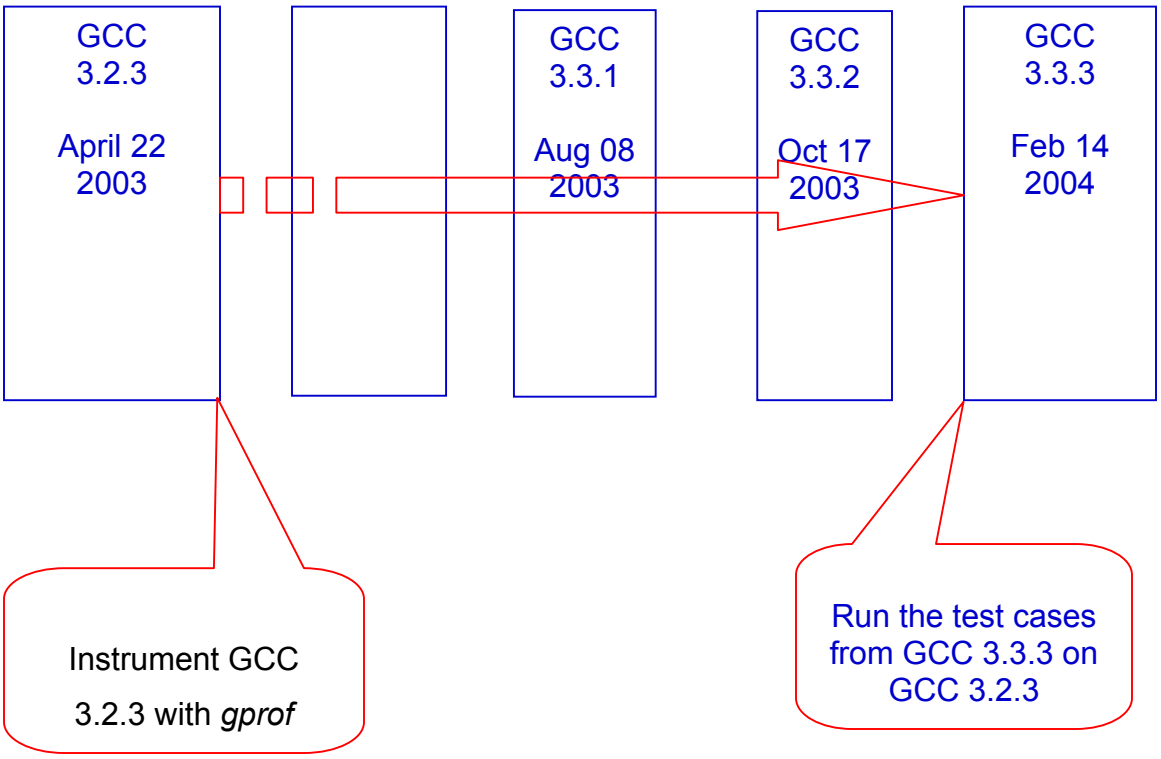


Figure 3.1: Experimental Setup

When we tested the regression test suite of GCC 3.2.3 against GCC 3.2.3, only 52 test cases failed out of the 21,000 test cases. However, 110 test cases failed out of the 2126 test cases chosen from GCC 3.3.3 when tested against GCC3.2.3. The reason is that, most of these test cases were added to the regression test suite of GCC 3.3.3 after GCC 3.2.3 was released. For example, the test cases that test the faults in GCC 3.2.3 are added to the test suite after the release of GCC 3.2.3. This is the reason behind choosing the test cases

from GCC 3.3.3 to test GCC 3.2.3, which will give us a better chance to locate the bugs in GCC 3.2.3.

The 21,000 test cases in the regression test suite of GCC 3.3.3 are arranged in 13 different folders. Each folder contains a unique set of test cases. Some of the folders contain test cases that test one particular language. GCC test suite has test cases to test C, C++, Java, Objc, Ada, and g77 front-ends. For example the folder *g++.dg* has test cases that test *C++* language part of GCC. As mentioned in section 3.2 we are only concerned with the *C proper* part of GCC. There are three sets of test cases that were written to test the *C proper* part of GCC. These are *gcc.c.torture, gcc.dg and gcc.misc.tests*. There are 2126 test cases in these three folders. All these test cases are written in the C language.

Test cases that run on any target machine are in *gcc.c.torture.* There are three subdirectories in *gcc.c.torture.* Test cases that merely need to be compiled are in *compile* directory, test cases, which should give an error are in *noncompile* directory and the execution tests are in *execute* sub-directory. All the test cases in *gcc.dg* are named against the feature of GCC that they test. For example all *bit-field* tests are named *bf-*.c.* The `gcc.misc.tests` folder has miscellanious test cases. Some of them are target specific and some of them test the profilers that come with GCC [1]. Different kinds of test cases can be run with a single test driver, using Dejagnu, a GNU testing tool.

Chapter 4

Experimental Setup

4.1 Introduction to profiling

We are using the *white-box* approach to extract the software failure behavior. In *black-box* testing we use a set of test cases to test the software. We estimate the reliability based on the test results with out considering the execution details. With black-box approach, we would not know why a test case was failed. In the *white-box* approach we also consider the execution details. We instrument the source code using a profiler to analyze the software executions.

Profiling is "the strategy of collecting calls, counts and execution times on a per function basis" [4]. The profile generated at the function level can be called specifically as 'function level profile'. We can define profile as a data set that stores all the data that belongs to an execution. The profile contains a lot of data about the execution of the given program. It tells us where a program spent its execution time, which functions were executed during that period, and which functions are called from which other functions. We can find which functions are executed most, which functions are slower than expected, and which functions are called more or less often than expected. It will help us in finding the key areas where a rewriting could be considered to make the program perform better. One may think that this can be done just by inspecting the source code, however we can only find static information like software metrics with this approach. Though the profiler does not give us any information about the failures in software directly, it gathers the execution data automatically. The data includes the functions that were visited, execution time and the function calls. This data can be used to analyze the failure behavior of the software. Since it is an automated process it can be used with large

complex programs that are too difficult to analyze by inspecting the source. The information we get from code inspection will be static, where as the profile gives us the dynamic information about the executions. However, the profiler doesn't tell us where the execution starts or where it ends. The profiler tells us where the program spent its time. Details about the profiler information are explained in section 4.1.1 and 4.1.2.

The profiler we use for our experiments is *gprof*, a GNU profiler developed by Jay Fenlason and Richard Stallman [4]. This is an open source profiler. The *gprof* can be used with many languages including C, C++, Pascal and Fortarn77. We are doing all our experiments on LINUX (Debian) system. We also tried a profiler called *gcov*, but this gave us profiles at line and block level. It is hard to analyze the profiles at block level especially for huge programs like GCC. *Gprof* gives the profile at the function level, which was easy to analyze compared to the profiles at line or block level. We found *gprof* as a good match for our requirements. Profiling with *gprof* has the following three steps.

- **Compile and link the program with profiling enabled**
  It is the first step in generating the profiles. When we run the compiler we have to use the option '-pg' in addition to all the options we use for the compilation.
- ***Execute the program to generate the profile data***

  After the program is compiled for profiling, we need to execute the program to generate the data that is needed by the gprof, to get the profile information. We can run the program with the normal options we used before. The program may run a little slow as it has to generate some extra information during execution. The information that is generated by the profiler is mostly effected by the program input and the type and number of arguments given when running the program. The profile only gives information about those parts of the software that are active during the execution. The program writes the profiling information into a file called *gmon.out***.** This file will overwrite any file that is named *gmon.out,* but we can change the file name or make a back-up copy once it is created. The

file will be written properly only when the test case executed normally, that is, it exits from the `main` function or by calling `exit`.

- **Run gprof to get the profile data file**

  Once we execute the program and the `gmon.out` is generated, we have to run *gprof* to interpret the information that the `gmon.out` contains. There are two kinds of profiles we get from *gprof*; *flat profile* where the list of all functions that were active during the program execution are listed, and a *call graph* where the history of all function calls is specified. We can save these profiles into a file by redirecting the standard output. The default executable file is `a.out` and the default `profile-data-file` is `gmon.out`. We can also give multiple `profile-data-files` to get the summarized report from all the profiles.

4.1.1 Flat profile

The flat profile shows the names of the functions that were active during the execution and the time that was spent in each. A sample flat profile generated by gprof is shown in Figure 4.1. This sample is taken from the documentation of gprof [5].

If we look at the flat profile shown in Figure 4.1, we can observe that the functions are ordered by the decreasing amount of time spent in them. There are some functions listed in the flat profile like `profil` and `mcount`, which were used for profiling itself. The time spent in them is the overhead that profiler brings into the execution.

```
Each sample counts as 0.01 seconds.
  %        cumulative    self                self    total
 time      seconds       seconds    calls    ms/call ms/call name
 33.34       0.02         0.02      7208      0.00     0.00  open
 16.67       0.03         0.01       244      0.04     0.12  offtime
 16.67       0.04         0.01         8      1.25     1.25  memccpy
 16.67       0.05         0.01         7      1.43     1.43  write
 16.67       0.06         0.01                              mcount
  0.00       0.06         0.00       236      0.00     0.00  tzset
  0.00       0.06         0.00       192      0.00     0.00  tolower
  0.00       0.06         0.00        47      0.00     0.00  strlen
  0.00       0.06         0.00        45      0.00     0.00  strchr
  0.00       0.06         0.00         1      0.00    50.00  main
  0.00       0.06         0.00         1      0.00     0.00  memcpy
  0.00       0.06         0.00         1      0.00    10.11  print
  0.00       0.06         0.00         1      0.00     0.00  profil
  0.00       0.06         0.00         1      0.00    50.00  report
```

Figure 4.1: Example of Flat profile

The meanings of the fields in Figure 4.3 are explained here.

- **% Time** is the percentage of the total execution time that the program spent in this function. These should all add up to 100%.
- **Cumulative seconds** is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one.
- **Self-seconds** is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number. This time is calculated using the sampling method. The sampling time is given at the starting of the flat profile.

- **Calls** is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the *calls* field is blank.
- **Self-ms/call** represents the average number of milliseconds spent in the function per call, if this function is profiled. Otherwise, this field is blank.
- **Total ms/call** represents the average number of milliseconds spent in the function and its descendants per call, provided that the function is profiled. Otherwise, this field is blank.
- **Name** is the name of the function. This field sorts the flat profile alphabetically after the ***self seconds*** field is sorted

The first line in the flat profile indicates the sampling time (0.01 seconds in this case) that is used to calculate the time periods for the function executions. If the time spent in the function is not considerably greater than the sampling time period it is considered as invalid. The sampling period estimates the margin of error in the time column. The program is monitored every 0.01 seconds. A time period of 0.01 seconds is assigned to the function that is active at that time. The function is assigned 0.02 seconds if it appears again. The last value in the 'cumulative seconds' column field tells the total execution time which is 0.06 in this case. That means only six samples are taken during the execution. One during the time when the execution is in 'open' and one for the *offtime*, *memccpy*, *write*, and *mcount*. *Self-seconds* tells how much time is spent in each function.

There are some functions like tzset, tolower and *strlen*, which have a non-zero value in the *calls* field but have a zero in the *self-seconds* field. The call graph (see section 4.1.2) is showing that these functions are called during the execution, but the time spent in them is shown here as zero. This indicates that the time spent in those functions is much less than the sampling time 0.01 seconds. So the profiler could not extract the time period for those functions due to the paucity of the histograms that were generated

[4]. As the number of samples taken is too small (6 in this case), none of these numbers in the *self seconds* column can be regarded as reliable. If we run the program again there is a possibility that we get different values for them [5]. Due to the *Statistical Sampling Error* (see section 4.1.3) none of these values are accurate.

4.1.2 Call Graph

The call graph contains entries for all the functions that are invoked during the execution. The call graph tells us how much time was spent in each function and its children functions during the execution. There may be some functions that have a very small execution time, but they call functions that use a significant amount of time. Figure 4.2 shows a call graph taken from the same profile as the flat profile example in Figure 4.1.

The dashed lines divide the table into entries for different functions. Each block represents one function. Each entry, corresponds to a function, which is identified by the primary line and starts with an index number in square brackets. The name of the function is at the end of the line. The preceding lines of the primary line describe the callers of the function (i.e. parents) and the following lines describe the descendants (i.e. children). All these entries are sorted by the total amount of time spent in them and their children. Unlike the flat profile, the functions that are used solely for profiling are not mentioned in the call graph. The fields in the call graph have different meanings in different contexts. Each context and the meanings of the fields are described in the following section.

Granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

```
Index % time    self  children    called     name
                                                <Spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1         main [2]
                0.00    0.00       1/2         on_exit [28]
                0.00    0.00       1/1         exit [59]
-----------------------------------------------------------------
                0.00    0.05       1/1         start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1         report [3]
-----------------------------------------------------------------
                0.00    0.05       1/1         main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8         timelocal [6]
                0.00    0.01       1/1         print [9]
                0.00    0.01       9/9         fgets [12]
                0.00    0.00      12/34        strncmp <cycle 1> [40]
                0.00    0.00       8/8         lookup [20]
                0.00    0.00       1/1         fopen [21]
                0.00    0.00       8/8         chewtime [24]
                0.00    0.00       8/16        skipspace [44]
-----------------------------------------------------------------

[4]     59.8    0.01        0.02       8+472    <cycle 2 as a whole>
                0.01        0.02     244+260      offtime <cycle 2>
[7]
        0.00      0.00    236+1       tzset <cycle 2>  [26]
-----------------------------------------------------------------
```

Figure 4.2: Call graph

4.1.2.1 The Primary Line

The primary line describes the function that the block belongs to. It has an index at the beginning and the name of the function at the end of the line. Figure 4.3 shows a primary line from the call graph.

```
Index   % time     self    children     called       name

[3]       100.0    0.00    0.05         1            report [3]
```

Figure 4.3: Sample from the call graph

The meanings of the fields in Figure 4.3 are explained here.

- **Index** is a unique number that is given to each function name at the beginning of its primary line. This number is used as an index for the function. When ever the function in primary line is used as a caller or a subroutine (child) this index is used along with its name.
- **% Time** is the percentage of the total time that was spent in this function. This includes the time spent in its children. The time for this function is added with its callers, so adding the percentages of time for its parents is meaningless to find *%time* here.
- **Self** is the total amount of time spent in this function. This is equal to the 'self seconds' field entry for this function in the flat profile.
- **Children** is the total time spent by the children of this function. This should be equal to the sum of all the 'self' and 'children' field entries for its subroutines.
- **Called** is the total number of time the function is called. There can also be recursive calls. If there are recursive calls this filed is represented as two numbers separated by a '+'. The first one represents the number of non recursive calls and the second one represents the recursive calls.
- **Name** is the name of the function, followed by the index number.

The cycles in the execution are named by the word *cycle* and they are represented by consecutive integers. If the function is part of a cycle, the cycle number is printed between its name and the index number. For example the function `offtime` is a part of the cycle 2 and has index number 7. So the primary line will have `offtime <cycle 2> [7]` at the end.

4.1.2.2 Function's callers

The functions that are listed above the primary line of the function are the callers of that function. They have the same fields as the primary line. But they have different meaning in this context. Figure 4.4 shows part of the call graph, which depicts the primary line of the function 'report' and its caller and a subroutine.

```
Index   % time     self     children     called          name

                   0.00     0.05         1/1             main [2]
[3]     100.0      0.00     0.05         1           report [3]
                   0.00     0.03         8/8             timelocal [6]
```

Figure 4.4: Sample from Call Graph

The meanings of the fields in Figure 4.4 are explained here.

- **Self** is the percentage of time spent in the function 'report' when it is called by 'main'.
- **`Children`** `is` the percentage of time spent in the subroutines of 'report' when it is called by 'main'. The sum of these two fields (self and children) is the percentage of time spent within 'report' when it is called by 'main'.
- **`Called`** is a combination of two numbers separated by a '/'. The first one is the number of times the function 'report' was called by 'main' and the second one is the total number of non-recursive calls to 'report' from any of the functions.

- **Name and index number** are the name of the caller function and the index number. Some times the caller function may not have its own entry. In that case there will be no index number after the name. If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number. The word 'spontaneous' appears in the **name** field if the caller has no identity.

4.1.2.3 Function's Subroutines

The lines that are below the primary line represent the subroutines of the function. Figure 4.5 depicts a small part of call graph that shows the function 'main' and its subroutines.

```
Index   % time    self    children   called     name
...

[2]     100.0     0.00    0.05       1          main [2]
                  0.00    0.05       1/1           report [3]
```

Figure 4.5: Sample from the Call Graph

The fields are same for both the *primary line* and the *subroutine line*. But they have different meaning in this context.

- **Self** is the amount of time spent within the function 'report' when it is called by 'main'.
- **Children** is the amount of time spent in the subroutines of 'report' when it is called by 'main'.
- **Called** is a combination of two numbers separated by a '/'. The first one is the number of times the function 'report' was called by 'main' and the second one is the total number of non-recursive calls to 'report' from any of the functions.
- **Name** is the name of the subroutine function followed by its index number. If it is the part of any cycle then the cycle number is also printed between the name and the index number.

4.1.2.4 Mutually recursive calls:

The output of *gprof* is very complicated to analyze because it considers cycles among functions. A cycle exists in the call graph if a function calls another function that calls the original function. But there is a problem with *gprof* regarding the cycles. If function **a** and function **b** call each other and **b** and **c** call each other all three functions belong to the same cycle. Even, when function **b** calls **a,** but **a** does not call **b,** *gprof* still considers it as a single cycle. However, the cycle information is not important for our research.

4.1.3 Statistical Sampling Error

Gprof uses sampling to find the time periods, so the time periods in the profiles have some statistical inaccuracy in them. For *n* samples the error rate is square root of *n*. For example if the sample time is 0.01 seconds and the total run time is 1 second then there are 100 such samples and the error rate is 10. If there is a function that has a very small amount of execution time so that the sampling can find that function only once, the profiler may find it zero times or even twice in some cases. The numbers are considered reliable only when they are much higher than the sampling time. The small numbers in the 'self seconds' tells us that these functions share an insignificant percentage of total time, so they need not be optimized.

Reducing the sampling period would give us more accurate values for the *self-seconds*. But unfortunately the sampling rate is not controlled by gprof itself. Instead it is handled by the special function `monstartup`, which is called by a profiled program when it starts up. This function uses the Linux operating system function *profil* to set up the time-based sampling. On typical UNIX systems, as well as on GNU/Linux, the precision of the gprof timer is determined by the behavior of the *profil* function. On GNU/Linux, *profil* is part of *glibc*, not a kernel system call. If we want to change the sampling time we have to find the sources for the *profil* function in the kernel and then examine them to see if you can change the sampling frequency. Then, we have to rebuild the kernel with this

changed function in it before we can start generating profiles with greater sampling frequency.

4.2 Profiling GCC test cases

4.2.1 Building the GCC sources and run test cases

All our experiments were performed on *GCC-3.2.3*, which was released on April 22, 2003, and *GCC-3.3.3*, which was released on February 14, 2004. Although we are concerned only about the compiler proper *cc1* (see Section 3.2), we downloaded the complete *GCC* with different language front-ends, so that we would not have any problems in building the *GCC* source. The sources for these versions are downloaded from one of the mirror sites of *GCC*. The details of the source code are explained in Section 3.2. Like any software from GNU, *GCC* needs to be configured before it can be built. To instrument the source code and generate the profiles for *GCC* some special options must be used while installing *GCC*. Some key differences between normal installation and profiling-enabled installation are:

- We need either CC or GCC added to the Unix PATH variable
- Have a separate directory for building the source code
- Use option "-g -profile -O2" while configuration
- Use "make all" instead of "make bootstrap"

As explained in Section 3.4 we used the test suite of GCC 3.3.3 to test GCC 3.2.3. We ran the test cases with a GNU testing tool called *dejagnu. Dejagnu* is a collection of *Tcl* scripts crafted to develop a test infrastructure that supports a specific tool [1]. There is no limit on the number of test cases that can be tested with *dejagnu*. *Dejagnu* is written in *expect*. *Expect* uses *Tcl*, a tool command language. *Dejagnu* is open source software developed by GNU.

Out of 20,000 test cases available with GCC 3.3.3 test suite we used only 2126 test cases, which are in the gcc.c.torture, gcc.dg and gcc.misc.tests folders because we are only testing the *C proper* part of GCC (see Section 3.4). We run all the test cases using a single *make* command. This tests all components of gcc, the language front-ends and all the runtime libraries. We can even run these test cases separately. We have separate test drivers written in *expect* to run each set of test cases separately. Details of these test cases are given in Section 3.4. GCC has targets *make-check-gcc* and *make-check-g++* which are used to test c and c++ language frontends separately. We can also run the test cases in different order by writing scripts in *expect*. When we run the test cases using the *make-check* command various **.sum* and **.log* files are created in the subdirectories of the testsuite [1]. The *.log files contain a detailed description of testcase name, type of testcase and the corresponding result. The *.sum summarize all the results. The results in the *.log files and .sum files are associated with the status codes shown in table 4.1. a sample from the *sum* (.sum) file is shown in Figure 4.6.

Status Code	**Meaning**
PASS	The test passed as expected
XPASS	The test unexpectedly passed
FAIL	The test unexpectedly failed
XFAIL	The test failed as expected
UNSUPPORTED	Test is not supported on this platform
ERROR	Testsuite detected an error
WARNING	The testsuite detected a possible problem

Table 4.1: Status codes used for log files in Dejagnu testing tool

It is very easy to interpret the results once we have the *log* files that are generated during testing. We considered both XFAIL and FAIL status codes as failures and we neglected those with the status codes UNSUPPORTED, ERROR. UNSUPPORTED and ERROR

status codes are generated due to the failures in running the test cases, for example failure in the test driver (Dejagnu).

```
PASS: gcc.c-torture/compile/20030821-1.c,  -O3 -fomit-frame-pointer
PASS: gcc.c-torture/compile/20030821-1.c,  -O3 -g
PASS: gcc.c-torture/compile/20030821-1.c,  -Os
FAIL: gcc.c-torture/compile/20030907-1.c,  -O0
FAIL: gcc.c-torture/compile/20030907-1.c,  -O1
FAIL: gcc.c-torture/compile/20030907-1.c,  -O2
PASS: gcc.c-torture/compile/981007-1.c,  -Os
XFAIL: gcc.c-torture/compile/981022-1.c,  -O0
XFAIL: gcc.c-torture/compile/981022-1.c,  -O1
XFAIL: gcc.c-torture/compile/981022-1.c,  -O2
……
…….
=== gcc Summary ===

# of expected passes            19903
# of unexpected failures        1355
# of expected failures          68
# of unresolved testcases       58
# of unsupported tests          100
/home/sunil/gcc/config/gcc/xgcc version 3.2.3
```

Figure 4.6: Sample from *.sum file generated by Dejagnu

We also generated execution profiles for all the test cases so that we can have an insight into execution path. Saving each profile separately after the execution is tedious as there are 2126 such profiles. We combined the process of testing and profiling and automated the whole process using *awk* scripts. The test cases are basically C programs. Every time we run a new test case (running a test case is nothing but compiling a *C* program) a new *gmon.out* file (a file in which all the profile information is stored) will be created in the same folder. The script will run the profiler and save the profile so that the new *gmon.out* file will not replace it.

Generating profiles for test cases is not a trivial task because we built both *cc1* (compiler proper) and the GCC driver binary (*gcc*) with profiling. The *gcc* program is just a driver that parses the options we give with *gcc* (the command), and calls whatever subprograms needed to compile the program. The functions that are called by *gcc* driver include the preprocessor, the compiler proper (*cc1*), the assembler, the linker, and possibly other programs. Since we built both *gcc* driver and *cc1* for profiling, the *gcc-gmon.out* overwrites the *cc1-gmon.out,* but it is the *cc1-gmon.out* that we want. Irrespective of the input given to the compiler the profile for *gcc* driver remains same. We had two options to choose from to resolve this problem. One was to run the compiler proper (cc1) separately. The second one was to rebuild the exact same sources without profiling (normal bootstrap) and then, instead of installing them, we can to copy *myconfig/gcc/xgcc* over */root/install/bin/gcc*. Relinking the *xgcc* executable without -p/-pg will solve the problem. We ran the compiler proper separately and generated profiles only for cc1 driver. This is the most reasonable way we found, to generate the profile for the compiler proper cc1.

Each time a profile is generated, it is stored along with the test case in the test case sub-directory. We have 2126 profiles, one for each test case. Each profile has a flat profile and a call graph. Each flat profile has 700 to 1000 functions in it. The call graph had more functions because it shows all the children and parents of each function that was executed. There is lot of redundant data in the profiles generated by *gprof*. We parsed the

file to extract the information we needed. We care only about the unique function names in the profile and which function called which other functions and how many times. The profiles we have are similar to each other in the sense that, there are many common functions in all the profiles. This is because the test cases are too small that there can be significant difference between the two profiles. We choose these test cases to analyze the behavior of GCC because they test different parts of the compiler and we have an oracle that tells us which test cases are passed and which test cases are failed, so that we can trace the faults related to these failures.

4.2.2 Mapping from functions to components

The profiler gives us the execution information at the function level. On average we have more than 700 functions in each *flat profile* (see Section 4.2.1). We found that there are 1759 unique functions in 2126 profiles we have. We are using state based approach to build the architecture-based software reliability model [7] [8]. We can build a control flow graph from the profile data by considering each function as a state in the system. It is very unrealistic that we build a control flow graph with 1759 states in it. It is very difficult to estimate the reliability at the function level because we do not have fault information at function level. More over there will be a state explosion in the Markov chain if we use all 1579 functions as states. We reduced the number of states by mapping these functions to the corresponding files. We used *ctags* to map functions to files. *Ctags* is an open source software developed by GNU, which is used to extract different kinds of tags in a C program [26]. A tag can be anything from a simple variable to something more complex, like a structure. We found that these 1759 functions belong to 108 source files in GCC.

Building the operational profile for GCC at the file level is not trivial because we have 108 files that control flows between. We would need to make 108 states in our Markov chain. Instead, we decided to further reduce the number of states by mapping these files to components. This is very hard because the documentation available for GCC in their official website and all other accessible resources is very old. Further more. We had limited domain knowledge. We used some information about the passes of compiler given in [6], but only 65 files have been mentioned in this documentation. In [6] the compilation process is divided into series of steps (*passes*) and a few files are assigned to each pass. However, that information was not enough to divide the system into components. More than 50 files out of 108 files are missing from the documentation. We looked in to the source code to understand what each file does and assign that file to the appropriate component. We divided GCC into 17 components, which have different functionalities. Files are assigned to components based on their functionality. The components and the number of files in each component are given in table 4.2

Component Name	Comp. ID	# of files
Parsing	1	32
Tree Optimization	2	11
RTL Generation	3	26
Jump Optimization	4	4
CSE	5	4
GCSE	6	2
Loop Optimization	7	10
Register Allocation	12	11
Branch Processing	13	4
Final Pass	14	9
Library Files	15	21
Top Level Control	17	1

Table 4.2: Component Reliabilities

4.3 Building Database

Since the profiles are extremely big, it would be inefficient to store them in flat files (see Section 4.2.2). A database is more efficient and manageable than a flat file when we use the information repeatedly [10]. We stored all the information in the database, so that we do not have to parse the profiles every time we want some information from them. We used JAVA programming language and ORACLE relational database (with JAVA database connectivity) for the parsing and database development respectively. We have the following four tables in our database

- Profile_Names
- Profile_Data
- Functionstofiles
- Component_Data

4.3.1 PROFILE_NAMES

The first table in our database is *Profile_Names*. *Profile_Names* table has two attributes. First one is the *profile name* and second is *profilenum* (the index number). *Profilenum* is the primary key for the table. A small sample from *Profile_Names* is shown in Table 4.3. Values are taken from the profile number one, the profile for the first test case. The only use of this table is to assign unique index numbers for the profiles so that they can be used in the remaining tables.

PROFILE NAME	PROFILENUM
Wreturn-type	1651
Wreturn-type2	1652
Wshadow-1	1653
Wswitch-2	1654
Wswitch-default	1655
Wswitch-enum	1656
Wswitch	1657
Wunknownprag	1658
Wunreachable-1	1659

Table 4.3: Sample from the *Profile_Names* table.

4.3.2 PROFILE_DATA

The second table in the database is *Profile_Data*. It contains function call data from all 2126 profiles. The data was taken only from the call graphs of all the profiles. *Profile_Data* is the most important table in the database because it has all the information from the profiles. All the remaining tables use information from *Profile_Data. Profile_Data* has five fields; *filenum, functionname, functioncalled, count* and *time*.

- ***Filenum*** is the index number created in the *Profile_Names* table.
- ***Functionnam*** is the name of a unique function in the profile.
- ***Functioncalled*** is one of those functions that were called by the *functionname* in the current profile.
- ***Count*** is the number of times the *functionname* called the *functioncalled*.
- ***Time*** is the time spent in the *functionname* in the current profile.

A sample from *Profile_Data* table is shown in Table 4.3. The values in the table are not from the original profile, but illustrate how the values in the table are organized. As

shown in table 4.3, the function calls related to the first profile are listed first, followed by the function calls related to profile two and so on. We have information about 2126 test cases in our database. Table 4.4 is just a sample of the data we have in profile_data. We have 4,643,491 rows in this table. One can estimate the size of data we have and the complexity of the case study we are using, by looking at this table.

FILENUM	FUNCTIONNAME	FUNCTIONCALLED	COUNT	TIME
1	insn_default_length	constrain_operands	435	0.01
1	insn_default_length	get_attr_i387	45	0.02
1	insert_insn_on_edge	emit_insns_after	43543	0.05
1	shorten_branches	emit_insns_after	34	0.01
1	constrain_operands	find_reg_note	5	0.01
2	propagate_block	propagate_one_insn	45	0.02
2	propagate_block	compare_tree_int	4354	0.01
2	size_diffop	compare_tree_int	356	0.01
2	bitmap_copy	propagate_one_insn	77	0.06
2	bitmap_copy	Bitmap_print	5	0.04
3	reg_to_stack	dead_or_set_p	6	0.05
3	reg_to_stack	find_regno_note	5657	0.01
3	reg_fits_class_p	try_split	67688	0.02
3	try_split	set_noop_p	678	0.04
3……	find_reloads	rtx_equal_p	676	0.01
…….	…….	…….	…….	…….
2126	Recog	push_operand	343	0.01
2126	Recog	Binary_fp_operator	33	0.01
2126	get_insn_name	immediate_operand	1	0.02
2126	make_insn_raw	ix86_binary_operator_ok	5547	0.01

Table 4.4: Sample from P*rofile_Data* table from our database

4.3.3 FUNCTIONSTOFILES

We divided GCC into components so that the data becomes more manageable (see section 4.2.2). *Functionstofiles* table has the mapping from functions to files and to components. We have three fields *functionname, filename* and *componentname* in this table.

- ***Functionname*** is name of the function,
- ***Filename*** is name of the file that *functionname* belongs to.
- ***Componentname*** is the component that the *filename* belongs to.

We have 1759 records in the table, one for each unique function. Table 4.5 shows a sample from the table *Functionstofiles.*

FUNCTIONNAME	FILENAME	COMPONENTNAME
error_with_file_and_line	diagnostic.c	Final Pass
gen_split_1038	gen.c	RTL generation
error_module_changed	Diagnostic.c	Final Pass
Record_last_error_module	Diagnostic.c	Final Pass
htab_hash_string	hashtab.c	System Library
in_data_section	varasm.c	Final Pass
set_named_section_flags	varasm.c	Final Pass
default_section_type_flags	varasm.c	Final Pass
Named_section_flags	varasm.c	Final Pass
default_elf_asm_named_section	varasm.c	Final Pass

Table 4.5: Sample from *Functionstofiles* table from our database

4.3.4 COMPONENT_DATA

C*omponent_Data* has the information about the profiles at the component level. C*omponent_Data* table has same structure as the *Profile_Data* table. The records are generated by combining information from *Profile_Data* and *Functionstofiles* tables. The function names in *Profile_Data* were replaced by the corresponding component names. We used the table *Functionstofiles* for the mapping. A small part of the table is shown in table 4.6. We have five fields in this table: *Filenum, Component Calling, Component Called, Count and Time.*

- ***Filenum*** is the index number assigned to the profile.
- ***Component Calling*** is the component where the calling function is in.
- ***Component Called*** is the component where the function called is in.
- ***Count*** is the number of times the *Component Calling* called the *Component Called*.
- ***Time*** is the time spent in the Component Calling.

FILENUM	COMPONENT CALLING	COMPONENT CALLED	COUNT	TIME
1	Parsing	Parsing	435	0.01
1	Jump optimization	RTL generation	45	0.02
1	Parsing	System Calls	43543	0.05
1	Tree optimization	Parsing	34	0.01
1	RTL generation	Jump optimization	5	0.01
2	Tree optimization	Parsing	45	0.02
2	Parsing	System Calls	4354	0.01
2	Tree optimization	RTL generation	356	0.01
2	Parsing	System Calls	77	0.06
2	Tree optimization	Parsing	5	0.04
3	RTL generation	RTL generation	6	0.05
3	RTL generation	System Calls	5657	0.01
3	Parsing	Parsing	67688	0.02
3	Jump optimization	System Calls	678	0.04
3…	Tree optimization	RTL generation	676	0.01
…….. …….	…….. …….	…… ……	…….. ……..	…… …….....
2126	Parsing	System Calls	343	0.01
2126	RTL generation	Jump optimization	33	0.01
2126	Parsing	Parsing	1	0.02
2126	Jump optimization	Parsing	5547	0.01

Table 4.6: Sample of *Component_Data* table in our database

Chapter 5

Building the Operational Profile

We gathered all the information needed to build the operational profile of GCC using *gprof* during the testing (see Section 4.2). The validity of the operational profile is very difficult to estimate because it requires an in depth knowledge of the field usage of the software. The usage of the software components differs from one execution to the other. Some components could be activated only by a very complex sequence of instructions whose frequency is very hard to estimate a priori [20]. We tried to build an operational profile for the system that closely reflects the actual behavior on a given system architecture. We used the regression test suite provided with the GCC source code. Details about the test cases are given in Chapter 3. These test cases were written to test different features like language specific constructs, variable declarations and memory allocation of GCC. We can also generate different operational profiles by running a subset of test cases. We are building the operational profile for the C proper part of GCC so we have chosen a specific set of test cases from the suite that are written in C (see Section 4.2.1).

We run these test cases using a tool called *dejagnu* (see Section 4.2). We used *gprof*, a GNU profiler to get the traces of the execution (see Section 4.1). As explained in section 4.3, we have developed a database to store all the data from the execution profiles. All the information we need to build the operational profile is in the database. We extracted the values from the database to find the transition probabilities of the components. We have 17 components in our system. We added two hypothetical states START and END, the starting state and an absorbing state respectively. START and END do not contain any files. They are just dummy states added to the system to complete the Markov chain [20]. Table 5.1 has the counts (number of functions calls) for the component interactions.

	S	1	2	3	4	5	6	7	12	13	14	15	17	END
start														
1		0	17035	4346470	0	0	0	939	152030	0	5851420	19668190	124999	0
2		1082149	0	252345	30757	0	0	0	148730	0	0	1247029	4	0
3		13404172	26008	0	981940	149491	211719	290	1021693	0	620596	21252895	22831	0
4		61536	188207	581490	0	0	0	44789	0	0	32594	0	0	0
5		2426	0	545	0	0	0	0	0	0	0	2426	0	0
6		616740	52200	192786	6	0	0	0	0	0	0	213159	0	0
7		67265	1117305	316213	352601	2426	0	0	0	0	21986	0	0	0
12		1959751	74365	6802282	215284	0	0	0	0	0	203071	23057510	617796	0
13		10600	1659	123313	845	0	0	23343	567	0	553	467	0	0
14		1637755	24892	1602131	0	0	0	0	2	0	0	24924764	14849	0
15		24309727	31238	5009164	49283	0	0	0	2.1E+08	5224	6326761	0	2426	0
17		148394	12025	178087	59388	0	5220	12866	56678	10440	1281859	89218	0	21664
END														

Table 5.1: Call counts for components in GCC

If X and Y are two states in the system, the entry (X, Y) in the table represents the number of times component X called component Y. These counts take all (2126) test cases into consideration. There are no entries for components 8, 9, 10 and 11 because no execution profile contains any function from the files in these components. The test cases we chose may not need these components to be invoked to finish the execution. We did not consider those calls that are from some function in the component to some other function in the same component. So we have zeroes for all (X, X) entries. We represented the operational profile, in the form of a graph shown in Figure 5.1.

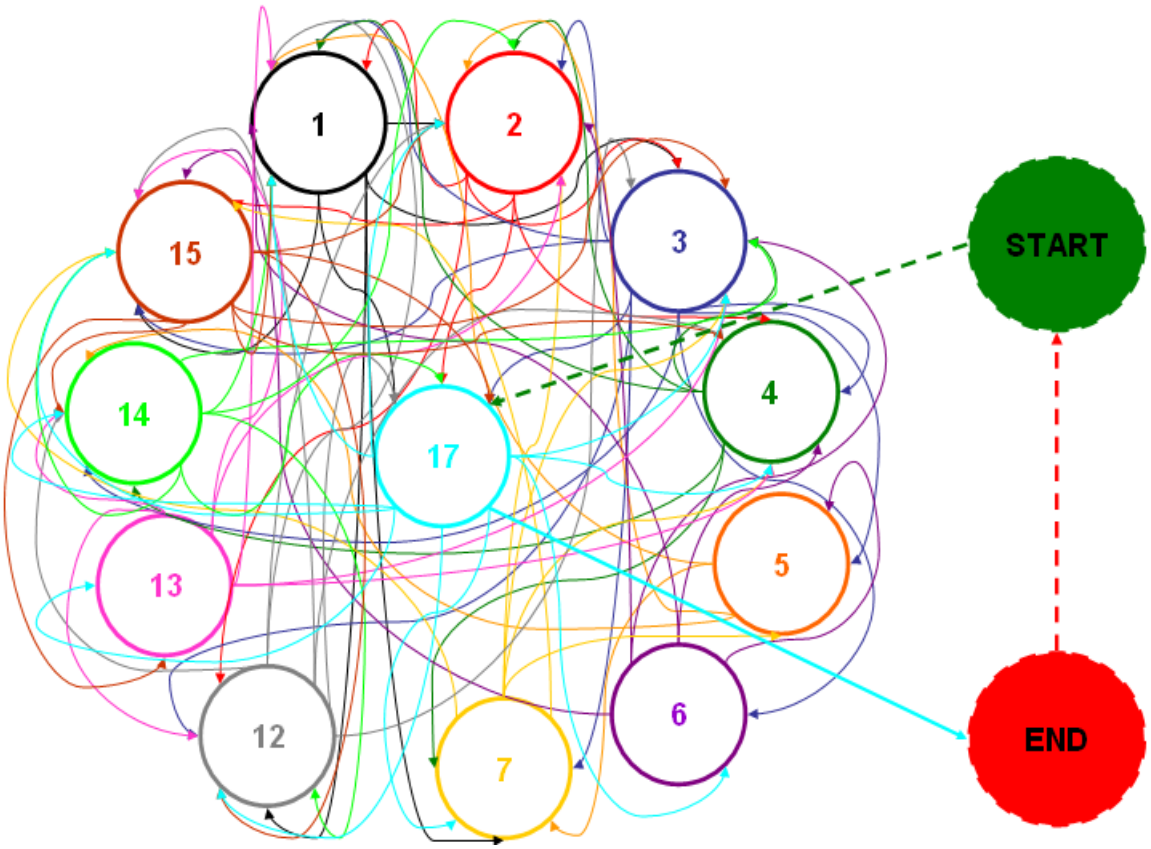


Fig 5.1: Operational profile of GCC

The component we have at the center (component 17) controls the execution. The compilation process starts and stops in component 17. We have a dummy state START from which the control transfers to component 17. Once the control goes to component 17, it handles calls to all the other passes and finishes the execution. We also have a state called END where the execution ends. The END component does not contain any files in it. It is a dummy state, which is used to represent the end of execution. In reality the execution starts and stops in component 17 itself. We can see the control passes from 17 to all other components.

The profiles generated from *gprof* have information about the functions that were visited and the number of times each function called other functions in the profile. They do not have information about the sequence of execution. It is very hard to find how many times the execution ends in component 17 because, we have multiple end points in component 17 as the assembly code for each and every construct of program will be generated separately by different functions in component 17 (in *toplev.c*). We had to find out manually which functions in component 17 will lead to an end.

The reason for multiple end points for GCC is *test case minimization* [1]. "A simplified test case means the simplest possible test case that still reproduces the bug. If you remove any more characters from the file of the simplified test case, you no longer see the bug"[11]. A test case is a C program in our experiments. Minimization is removing the part of the program that does not test the program and keeping only the part of the program that causes the failure of the system. A minimized test case may not be a complete C program but just a part of it. There are different functions in the *toplev.c* that take care of different constructs of C program. We needed to consider all such functions to come up with our hypothetical END state.

The transition probability matrix is shown in table 5.2. The entry in cell (X, Y) represents the probability that component X calls component Y. The values in Table 5.2 are derived from Table 5.1 using the Equation 5.1.

$$P_{ij} = \frac{n_{ij}}{n_i} \tag{5.1}$$

Where $n_{ij} = \sum_j n_{ij}$

	S	1	2	3	4	5	6	7	12	13	14	15	17	END
S	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1		0	0.0006	0.14411	0	0	0	3.11E-05	0.00504	0	0.19401	0.652105	0.00414	0
2	0	0.39194	0	0.0914	0.01114	0	0	0	0.05387	0	0	0.451656	1.45E-06	0
3	0	0.35563	0.0007	0	0.02605	0.004	0.00562	7.69E-06	0.02711	0	0.01647	0.563862	0.00061	0
4	0	0.06772	0.2071	0.63997	0	0	0	0.049294	0	0	0.03587	0	0	0
5	0	0.44951	0	0.10098	0	0	0	0	0	0	0	0.449509	0	0
6	0	0.57377	0.0486	0.17935	5.58E-06	0	0	0	0	0	0	0.198308	0	0
7	0	0.03582	0.595	0.1684	0.18777	0.0013	0	0	0	0	0.01171	0	0	0
12	0	0.05951	0.0023	0.20657	0.00654	0	0	0	0	0	0.00617	0.700196	0.01876	0
13	0	0.0657	0.0103	0.76427	0.00524	0	0	0.144676	0.00351	0	0.00343	0.002894	0	0
14	0	0.05807	0.0009	0.0568	0	0	0	0	7.09E-08	0	0	0.883719	0.00053	0
15	0	0.09972	0.0001	0.02055	0.0002	0	0	0	0.85341	2.14E-05	0.02595	0	9.95E-06	0
17	0	0.07911	0.0064	0.09494	0.03166	0	0.00278	0.006859	0.03021	0.00557	0.68335	0.047562	0	0.012
END	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.2: Transition probability matrix for GCC

Chapter 6

Fault Detection

6.1 Mapping Failures to Faults

“”Failure is a departure of system behavior in execution from user requirements” [22]. Failure is a user-oriented concept in the sense that it must occur during the execution of the software by the user. The defects detected by source code and design inspections cannot be considered as failures. These defects may not cause a failure during the actual execution of the software. “Fault is the defect that causes or can potentially cause the failure when executed” [22]. Fault is developer oriented in the sense that it is generated because of an error during the development of the software. For example, suppose that a particular output is expected for a test case and it does not occur, it might be because of some missing code or some incorrect code in the software, which is a fault.

We encountered failures during the execution of the test cases. Now, we are trying to map these failures to faults in the software. Finding the number of times each component is executed is very easy because we have all the data in our database. We can directly get the values from the database using SQL queries. It is hard to find out how many times each component has failed. For all those test cases that failed, we had to find out why they were failed. Neither the log files, that are generated during the testing nor the execution profiles generated by gprof contain any information about the location of the fault. There is no documentation given by the GCC developers regarding the test case failures.

We employed the following three methods to find the failure information.

- Searching *test case Change-Logs* and *Source-code Change-Logs* of version 3.3.3
- Search the bug-tracking database Bugzilla
- Execute tests on newer versions & search logs

Section 6.1 explains each of the methods in detail.

6.1.1 Searching test logs & change logs of version 3.3.3

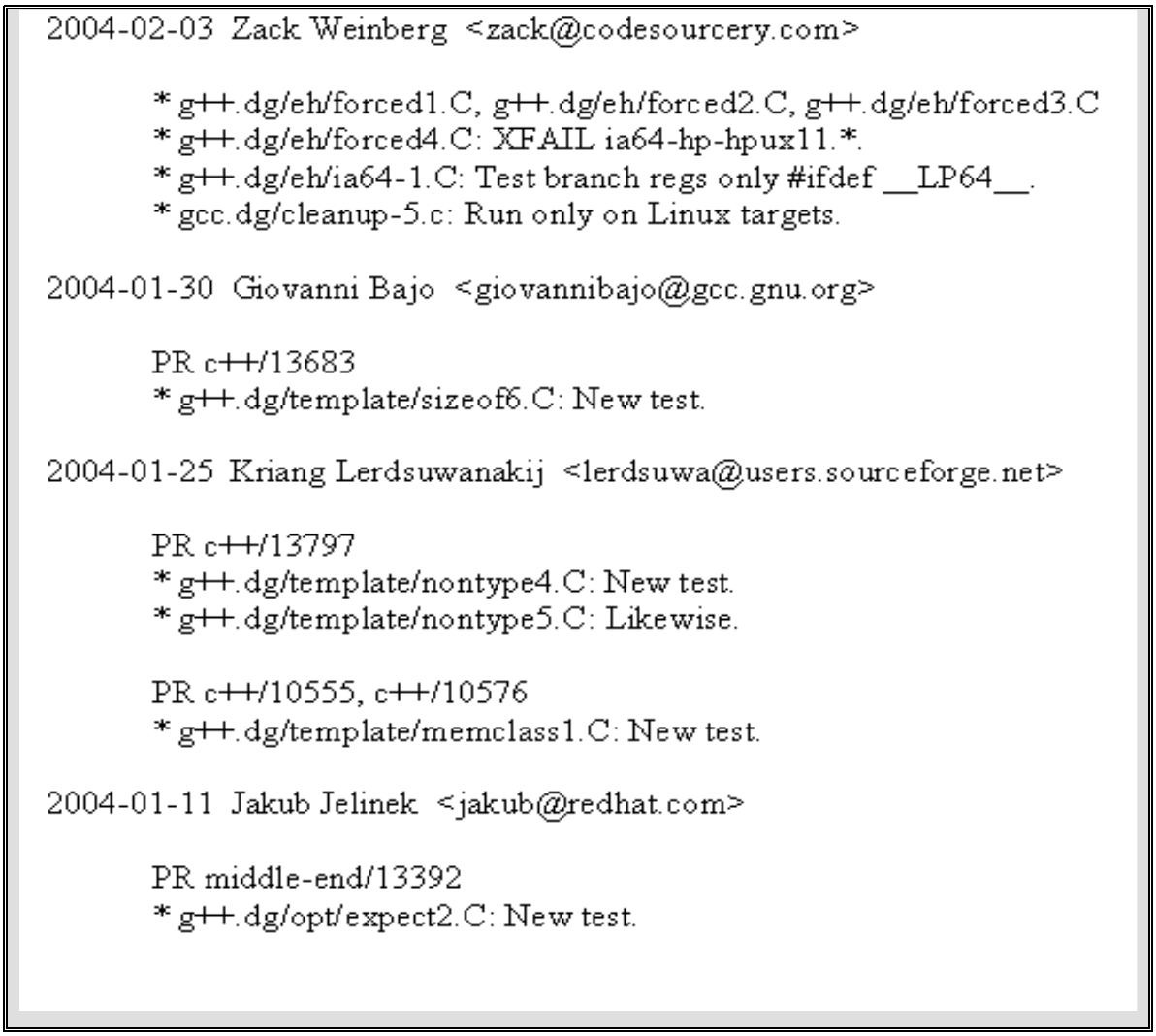
As we discussed in chapter 3, different versions of GCC are released periodically. Every version of GCC comes with a regression test suite. To track the changes made to the test suite, the developers maintain a database that has all the changes made to the test suite. The data is in the form of *Change-Log* files. *Change-Log* files are provided along with the test cases. There are two kinds of Change-Log files; *Source-code Change-Log* files and *test case Change-Log files*.

Test case Change-Log files contain the change information about the test suite that comes with GCC source code. They contain information about when a test case is added to the suite, who added that and what kind of test case it is. A sample from one of the log files is shown in Figure 6.1.

Source-code Change-Log files contain the change information about the source files in GCC. They are renewed with every version of GCC. The information in the *Source-code Change-Log* file includes when a file is changed, who changed the file and how the file was affected by the change. A sample from the file is shown in Figure 6.2.

In Figure 6.1, the first line in each entry has the date on which the test case has been written, name of the author, and the email address For example the line "2004-02-03 Zack Weinberg zack@codesourcery.com" tells us that Zack Weinberg has made a change to the *Change-Log* on 2004-02-03. The following lines have the test cases that were added by that author on that date. For example, the line "g++.dg/eh/forced4.C: XFAIL ia64-hp-hpux11" tells us that the `forced4.c` test case was added to *g++.dg* folder. The most important information here is the PR number (Problem Report number) given to each added test case. This PR number will be used as an index to search through the Bugzilla database (see Section 6.2). There are more than 3500 entries in the Test-case Change-Log file. However, we have only 169 PR numbers corresponding to the test cases

we used. That is why the searching Bugzilla was not so successful because the search was based on PR numbers.



```
2004-02-03  Zack Weinberg  <zack@codesourcery.com>

        * g++.dg/eh/forced1.C, g++.dg/eh/forced2.C, g++.dg/eh/forced3.C
        * g++.dg/eh/forced4.C: XFAIL ia64-hp-hpux11.*.
        * g++.dg/eh/ia64-1.C: Test branch regs only #ifdef __LP64__.
        * gcc.dg/cleanup-5.c: Run only on Linux targets.

2004-01-30  Giovanni Bajo  <giovannibajo@gcc.gnu.org>

        PR c++/13683
        * g++.dg/template/sizeof6.C: New test.

2004-01-25  Kriang Lerdsuwanakij  <lerdsuwa@users.sourceforge.net>

        PR c++/13797
        * g++.dg/template/nontype4.C: New test.
        * g++.dg/template/nontype5.C: Likewise.

        PR c++/10555, c++/10576
        * g++.dg/template/memclass1.C: New test.

2004-01-11  Jakub Jelinek  <jakub@redhat.com>

        PR middle-end/13392
        * g++.dg/opt/expect2.C: New test.
```

Figure 6.1: Sample from the Test Case *Change-Log*

The entries in the *Source-code Change-Log* (Figure 6.2) have the same format as the test case change log. The information here is different from the test case *Change-Log*. We can see the date and the name of the author followed by the list of files that were changed by that author on that date and a very brief description of the changes made. We used both test case *Change-Log* files and the *Source-code Change-Log* files in parallel to track the failures. In [12], [27], [28] Andy Podgurski, David Leon and William Dickinson used the

failure information about GCC test cases. However, the authors did not use the *Change-Log* files to find the faults. They run the test cases on different versions to find out when the test cases stopped failing. there are more than one *Source-code Change-Log* files in the GCC source code which are arranged chronologically. Each *Source-code Change-Log* has more than 19,000 entries. The ratio of PR numbers in *Source-code Change-Log* is even smaller than *Test-case Change-Log*. We have only 3550 PR numbers. this is the reason why we did not use PR numbers as a key for mapping changes from *Test-case Change-Log* to *Source-code Change-Log*.

```
2004-02-03  Wolfgang Bangerth  <bangerth@dealii.org>

        PR other/14003
        * doc/invoke.texi (x86 options): Fix spelling/wording.

2004-02-01  Geoffrey Keating  <geoffk@apple.com>

        PR bootstrap/13960
        * config/rs6000/rs6000.h (LEGITIMATE_LO_SUM_ADDRESS_P): Accept
        lo_sum addresses on Darwin.

2004-01-30  Eric Botcazou  <ebotcazou@libertysurf.fr>

        * config/sparc/sparc.c: Update copyright.
        * config/sparc/sparc.h: Likewise.
        * config/sparc/sparc.md: Likewise.

2004-01-29  Roger Sayle  <roger@eyesopen.com>

        PR java/13824
        * tree.c (unsafe_for_reeval): Handle EXIT_BLOCK_EXPR nodes specially
        as their EXIT_BLOCK_LABELED_BLOCK operands can lead to unbounded
        recursion.

2004-01-29  H.J. Lu  <hongjiu.lu@intel.com>

doc/invoke.texi: Remove the pni option from -mfpmath=.
```

Figure 6.2: Sample from *Source-code Change-Log*

We can track a failure and map it to the corresponding fault using these two files. When a test case is written, it is tested against the latest version available. If the test case fails, the author will make an entry in the *test case Change-Log*, along with the date and the author name. Then he tries to find out the location of the fault and fix it. Once the problem is fixed, an entry will be made in the *Source-code Change-Log* representing the date, author, and the files that were changed to fix the problem. We use this information to track the failures and map the failures to faults in the source code. For example, if we have a test case that failed on GCC 3.2.3, we search for the test case file name in the test case *Change-Log* (of GCC 3.3.3) first. Then we use the name and date of that entry to search the *Source-code Change-Log*, which when found tells us, all the files that were changed to ensure that the failure would not happen again. There can be other entries in the Change-Log files due to the changes made for the enhancements in the system. We find the defects at the file level. We could find 41 defects with this method. This is 75% of the total defects that we found. This is the most efficient method among the three. We automated the whole process using *awk* (a Unix scripting language). The script searches the *Test case Change-Log* and records the name of the person and the date of the change entry, and then searches the *Source-code Change-Log,* to find the entry with the same date and name. Since these two log files are in same format, it was very easy to automate the process.

6.1.2 Search the bug-tracking database Bugzilla

Bugzilla is a "Defect Tracking System" or "Bug-Tracking System" [13]. Bug tracking systems are used by developers to keep track of the bugs in the software. Bugzilla is a free software from *GNU* developers. It is most widely used bug tracking system not only because its free, but also for the features it has. Bugzilla is a powerful tool that will help a group of developers to get organized better and communicate effectively. Bugzilla also helps in reducing the downtime, increasing the productivity, and reducing project costs. Here are a few special features that bugzilla provides [13].

- Optimized database structure for increased performance and scalability
- Excellent security to protect confidentiality
- Advanced query tool that can remember your searches
- Integrated email capabilities and comprehensive permissions system
- Editable user profiles and comprehensive email preferences

Bugzilla contains information about bugs in software. Every bug entered in *bugzilla* database is given a set of attributes. Table 6.1 explains the attributes. *Bugzilla* provides an excellent query facility to search for bugs based on these attributes. A sample output of the b*ugzilla* query is shown in Table 6.2. Every bug has a detailed description associated with it.

The PR numbers extracted from the test case *Change-Log* files are used to search in the *description page* of a bug. This was a manual process, and was also the most inefficient method of the three proposed because there are no PR number entries for all the test cases in the *test case Change-Log* files. PR number is the only index that can be used for searching the bugzilla database. We found only 3 defects with this method, which is only 5% of the total bugs found.

Attribute	Purpose	Possible values
Bug-id	Unique ID given to the bug	**Any valid integer**
Status	Define and track the life cycle of a bug	**UNCONFIRMED** **NEW** **ASSIGNED** **WAITING** **SUSPENDED** **REOPENED** **RESOLVED** **VERIFIED** **CLOSED**
Resolution	Define and track the life cycle of a bug	**FIXED** **INVALID** **WONTFIX** **DUPLICATE** **WORKSFORME**
Severity	Describes the impact of a bug	Critical Normal Minor Enhancement
Priority	Describes the importance and order in which a bug should be fixed	P1 P2 P3

Table 6.1: Table that describes the bug attributes

bug_id	bug_severity	priority	Gcchost	short_short_desc
57	Normal	P3	-	confusing name lookup diagnostic
99	Normal	P3	-	Constant expressions constraints
100	Normal	P3	-	Statement expressions issues
157	Minor	P3	-	-
189	Normal	P3	-	-
192	enhancement	P3	-	-
336	enhancement	P3	-	-
346	Normal	P3	-	-
378	Normal	P3	-	-
429	enhancement	P3	i386-pc-linux-gnu	-
448	Normal	P3	i686-pc-linux-gnu	-
456	Normal	P3	i686-pc-linux-gnu	
529	Minor	P3		
545	Normal	P3	i686-pc-linux-gnu	
561	enhancement	P3	-	
576	enhancement	P3	-	
592	Normal	P3	-	
605	enhancement	P3	*-sun-solaris*	
704	enhancement	P3	i686-pc-linux-gnu	
712	normal	P3	-	
772	normal	P3	i686-pc-linux-gnu	
864	enhancement	P3	-	
914	normal	P3	i686-pc-linux	
950	normal	P3	-	

Table 6.2: Sample of *bugzilla* bug database

6.1.3 Execute tests on newer versions & search logs

We could not find the defect information about some of the test cases using either of the above methods. As a final bug location method, we tried to test those test cases on newer versions that were released after 3.2.3) and see where exactly it executed without a failure. We have three versions released between GCC 3.3.3 and GCC 3.2.3. This method is used by Andy Podgurski, David Leon in [12]. Even though we used this method as our final option, it proved to be more efficient than searching the *bugzilla* for defects. After identifying the version in which the bug was fixed, '*Searching test logs & change logs'* was repeated here to find the defects (as explained in section 6.1). We found 20 defects with this method, which is 20% of the total defects found.

6.2 Estimating component reliability

"The reliability of component ***i*** is the probability $\boldsymbol{R_i}$ that the component performs its function correctly" [20]. There are many methods to calculate the component reliabilities. We can use the historical data and the requirement documents if the project is in early stages. We can also use software reliability growth models for each component [18], [19]. However, the failure data available may not always sufficient to apply these models. Here we used information about the non-failed executions together with information about the failed executions during the testing to find the component reliabilities [14], [15], [16], [17]. These methods depend heavily on the type and nature of the test cases used to find the faults. Irrespective of the method used to find out the component reliabilities, the values may be inaccurate. We estimated the mean value of reliability for each component. Equation 6.1 gives the reliability of a component.

$$R_i = 1 - \lim_{n_i \to \infty} \frac{f_i}{n_i} \tag{6.1}$$

Where f_i is the number of failures and n_i is the number of executions of component i in N randomly generated test cases.

Table 6.3 shows the number of times each component was executed. We also found the number of times each component failed. After we found the defect information about all the components we calculated the component reliabilities using equation 6.1**.** The component reliabilities are shown in table 6.3.

Comp. ID	F_i	N_i	R_i
1	30	1,656,221	0.99998189
2	1	135,180	0.99999260
3	7	1,688,076	0.99999785
4	0	162,338	1.00000000
5	1	11,326	0.99991171
6	1	57,377	0.99998257
7	0	72,680	1.00000000
12	0	372,486	1.00000000
13	0	16,087	1.00000000
14	4	381,046	0.99998950
15	10	919,668	0.99998912
17	1	302,504	0.99999669

Table 6.3: Component Reliabilities

Comp_ID is a unique identification number given to the component. Component ID's 8, 9, 10, 11 and 16 are not shown because they were not executed by any of the 2126 test cases. F_i is the number of times the component failed in the 2126 test cases. N_i is total number of times the component is executed in 2126 test cases. R_i is the reliability of the component *i.* We can observe that the reliabilities are extremely high and almost equal to one. This is because we have few failures compared to the number of executions of each component.

6.3 Estimating system reliability

We used the state-based composite method proposed by R.C. Cheung in [21] to combine the software architecture with the failure behavior of the software. The model assumes a single entry node and single exit node for the system. We added two absorbing states C and F to the discrete time Markov chain (DTMC). These states represent the successful completion and failure of the system respectively. We already have added two dummy states START and END in the operational profile (Figure 5.2), which represent the beginning and ending of the execution. The transition probability P is converted to P_1. The transition probability Pij in the original matrix (Table 5.2) is converted to RiPij to generate the values in P_1. Ri is the reliability of the component *i*. RiPij is the probability that the component i produces the correct result and the control is transferred to component j [20]. An arc is made between the failure state and the component i with a transition probability of (1 - Ri), to consider the failure of component i. The components C and F are not considered when calculating the system reliability. The reliability of the system is the probability that the control reaches state C from START state. The matrix P_1 is converted into Q by deleting the rows corresponding to C and F. The element Q_k (1, n) represents the probability of reaching state n from START state with k transitions [20]. The number of transitions ranges from 0 to infinity. We can prove that $S = \sum_{k=0}^{\infty} Q^k = (1-Q)^{-1}$ [9]. So the system reliability is $R = S$ (1, n) R_n. We used MATLAB to implement the equation to find the system reliability.

The value for the reliability calculated using this method is 0.9201. We also found the reliability of the system using the black box testing and compared the two values. The error in the reliability estimate is given by equation 6.2. The error in estimation is a mere 5.5%.

$$\left|\frac{R_{model} - R_{actual}}{R_{actual}}\right| \cdot 100 = \left|\frac{0.9201 - 0.9741}{0.9741}\right| \cdot 100 = 5.5\% \qquad (6.2)$$

Chapter 7

Uncertainty analysis using entropy

We estimated the system reliability using the architecture-based methodology proposed in [20]. They derived an equation for the system reliability in terms of the transition probability P_{ij} and the component reliability R_i, however, there exists a considerable uncertainty in the software usage and failure of the components. There will be a certain amount of uncertainty in the reliability calculation even though the mathematical model is accurate [7]. We studied the uncertainty in the operational profile and the software reliability model. We used a method that is based on source entropy to analyze the uncertainty in the software reliability model [9]. This method can be used to assess the uncertainty of the operational profile and software reliability model.

Entropy is a very important concept in the field of information theory. In information theory entropy is used to estimate, to which extent a source can be compressed. Entropy calculates the amount of uncertainty in a Markov source. Equation 7.1 gives the entropy of the system.

$$H = -\sum_i \pi_i \sum_j p_{ij} \log p_{ij} \tag{7.1}$$

Here $\prod_i$ represents the steady state probability of state *i*. P_{ij} is the transition probability of the stochastic source. The range of the value is $0 \leq H(S) \leq \log(n)$ [37] where H(S) is the entropy of the system**.** The entropy for GCC is calculated as 1.0913. We also quantified the uncertainty of the components using the concept of conditional entropy. The uncertainty of the component *i,* is given by equation 7.2. The values of the uncertainties are shown in Table 7.1.

$$H_i = -\sum_j p_{ij} \log p_{ij} \tag{7.2}$$

Where P_{ij} is the transition probability

Component Name	Comp. ID	Expected Execution Rate	Component Uncertainty
Parsing	1	0.1007010	1.3418020
Tree Optimization	2	0.0023429	1.6623364
RTL Generation	3	0.1005563	1.4597536
Jump Optimization	4	0.0052616	1.5319009
CSE	5	0.0003992	1.3711187
GCSE	6	0.0005844	1.5794114
Loop Optimization	7	0.0003185	1.5911179
Register Allocation	12	0.3445442	1.2925663
Branch Processing	13	0.0000477	1.1466891
Final Pass	14	0.0386909	0.6457689
Library Files	15	0.3994340	0.7833594
Top Level Control	17	0.0070462	1.6905673

Table 7.1: component uncertainties

The table also has the values for estimated execution rates $\prod_i$ for all the components. The execution rates $\prod_i$ and the component uncertainties are shown in Figure 7.1. We can observe that component 1, 3, 12 and 15 have high expected execution rates compared to the other components. Components such as 5, 6, 7, and 13 have low expected execution rates compared to the others, which proves that the software executions are skewed. The components with a higher execution rate are expected to affect the system more than those that have a lower execution rate. The components with higher uncertainty will have greater impact on large part of the system.

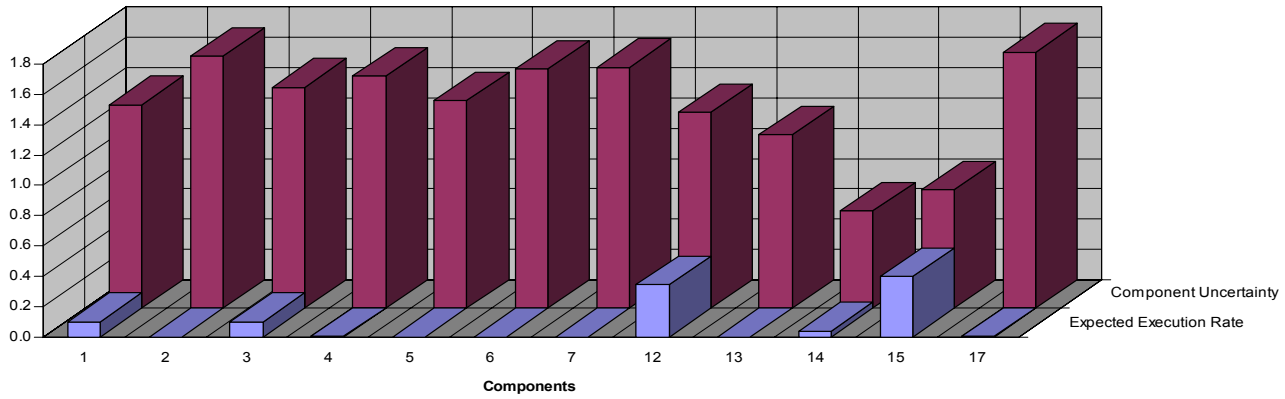


Figure 7.1: Expected execution rate and component uncertainty graph

We can observe that components 2 and 17 are more critical in the system because they have higher uncertainty. In Figure 7.1, components 12 and 15 have higher expected execution rates however; component 12 is considered to be more critical since it has a higher component uncertainty, which means that it will affect larger part of the system.

Chapter 8

Conclusion

This thesis presents the architecture-based reliability analysis of a large-scale open source application. We implement the Architecture-based methodology for uncertainty analysis of software reliability proposed in [20] to estimate the reliability of the system and to study the uncertainty analysis of reliability using entropy. We used GCC, a GNU open source compiler for our experiments. This is the biggest case study ever used for reliability analysis. The most important thing that differentiates our research from most of the related work is the magnitude of the case study we are using. The problems associated with experiments on empirical studies are explained. We addressed most of the potentially difficult problems associated with large-scale software applications. All previous studies on empirical studies mentioned in the related work, contributed to a small set of these problems. We presented an architecture-based methodology for reliability analysis. This methodology uses state based approach to find three important features of the software; the software-architecture, software-usage, and the software failure behavior, which are necessary to calculate the reliability of the system. An empirical approach for the architecture based reliability analysis was proposed, which uses white-box approach for the reliability analysis.

We used *informed-approach* to estimate the software architecture. The regression test suite provided with GCC 3.3.3, which has test cases to test GCC 3.2.3 and a testing tool called *Dejagnu* were used for testing GCC. G*prof*, a GNU open source profiler, was used to extract traces of the test case executions. The process of running the test cases and saving the profiles was automated. We used *call-graph* generated by gprof, to find the interaction of different components in GCC during the test case executions. However, the profiler gave execution profiles at function level. We mapped these functions to 108 files in the source code. These files were further mapped to 17 components in GCC. The source code of GCC was inspected manually to come up with the mappings from

functions to files and files to components. The profiles generated by *gprof* were huge and difficult to manage. A database was built to save the information from the profiles and to make the mapping easier. We extracted data from the database to build the operational profile for GCC.

We mapped the test case failures to the faults in the source code. *Test-case Change-Logs* and Source-Code Change-Logs, which were provided with GCC source, were used for this purpose. We implemented three different methods; *Searching test case Change-Logs and GCC Change-Logs of version 3.3.3, Search the bug-tracking database Bugzilla, Execute tests on newer versions & search logs*, to map the failures to faults in the source code. The first method proved to be most effective. We automated the whole process of searching through the change log files and mapping failures from faults, unlike other researches that used manual inspection.

The reliability is calculated for each component. The system reliability is calculated using both black-box method and the white-box method that we implemented. We got nearly accurate value for the reliability, with only 5% of difference between the values found using the two methods. The component uncertainty was analyzed using the method proposed in [9]. This method uses *entropy* as a measure of component uncertainty. Source entropy quantifies the uncertainty of the operational profile and architecture-based reliability models. We found the critical components that have high uncertainty value, which require more testing efforts than the other components. Further, the architecture-based methodology helps us to estimate the expected execution rate and uncertainty of each component using the theory of Markov chains and conditional entropy respectively.

In summery, the results presented in this thesis enrich the empirical knowledge in software reliability engineering. Lessons learned from this large-scale experiment are expected to be useful for conducting similar studies in the future.

References

[1] www.gnu.org

[2] http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

[3] http://www.delorie.com/gnu/docs/gcc/gccint_30.html

[4] http://www.gnu.org/manual/gprof-2.9.1/html mono/gprof.html

[5] http://www.cs.utah.edu/dept/old/texinfo/texinfo.html

[6] http://www.ictp.trieste.it/texi/gpp/gpp_55.html

[7] K. Goseva Popstojanova and K. S. Trivedi, "Architecture-Based Approach to Reliability Assessment of Software System", *Performance Evaluation*, Vol. 45, N0. 2-3, 2001, pp. 179-204.

[8] K. Goseva Popstojanova, A. P. Mathur, and K. S. Trivedi, "Comparison of architecture-Based Software Reliability Mod els", *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, 2001, Hong Kong, pp.22-31.

[9] S. K. Kamavaram and K. Goseva Popstojanova, "Entropy as a Measure of Uncertainty in Software Reliability", *Proc. 13th International Symposium Software Reliability Engineering, Supplementary proceedings* 2002,pp. 209-210.

[10] C. J. Date, Hugh Darwen: Relational Database Writings 1989-1991 *Addison-Wesley,* 1992.

[11] Andreas Zeller and Ralf Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", *IEEE Trans. Software Engineering,* Vol. 28, No. 2, February 2002.

[12] Andy Podgurski, David Leon, Patrik Francis, Wes Masri, Melinda Minch, Jiayang Sun and Bin Wang, "Automated support for classifying software failure reports", *Proc. 25th International Conference on Software engineering,* 2003, pp. 465 – 475.

[13] http://www.bugzilla.org

[14] B.Littlewood and D.Wright, "Some Conservative Stopping Rules for Operational Testing of Safety – Critical Software" *IEEE Trans. Software Engineering*, Vol.23, No.11, 1997, pp.673-683.

[15] K.W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nikol, B.W. Murrill, and J. M. Voas, "Estimating the Probability of Failure when Testing Reveals no Failures", *IEEE Trans. Software Engineering*, Vol.18, No.1, 1992, pp. 33- 43.

[16] E. Nelson, "A Statistical Bases for Software Reliability", *TRW-SS-73- 02, TRW Software series*, 1973.

[17] J.H.Poore, H.D.Mills and D.Mutchler, "Planning and Certifying Software System Reliability", *IEEE Software*, 1993, pp 88- 99.

[18] W. Everett, "Software Component Reliability Analysis", *Proc. Symp. Application–Specific Systems and Software Engineering Technology*, 1999, pp 204-211.

[19] S. Gokhale, W. E. Wong, K. Trivedi, and J. R. Horgan, "An Analytical Approach to Architecture Based Software Reliability Prediction", *Proc. 3rd Int'l Computer Performance & Dependability Symp (IPDS'98)*, 1998, pp. 13-22.

[20] K. Goseva Popstojanova and Sunil. K. Kamavaram, "Assessing Uncertainty in Reliability of Component-Based Software System", *Proc. 14th IEEE International Symposium on Software Reliability (ISSRE 2003)*, Denver, CO, Nov. 2003.

[21] R. C. Cheung, "A User-Oriented Software Reliability Model", *IEEE Trans. Software Engineering*, Vol.6, No.2, 1980, pp. 118-125.

[22] John Musa, "Software Reliability Engineering", McGrawhill

[23] Swapna S. Gokhale, W.Eric Wong, Kishor S.Trivedi and J.R. Horgan, "An analytical approach to Architecture-based software reliability prediction", *IEEE Internation Computer Performance and Dependability Symposium (IPDS'98)*, September 07 - 09, 1998.

[24] Thomas J.Ostrand and Elaine J. Weyuker, "Difficulties Encountered doing empirical studies in an industrial environment", *Proc. 15th IEEE International Symposium on Software Reliability (ISSRE 2004)*, Bretagne, France, Nov. 2004.

[25] M.Lipow, "Number of faults per line of code", IEEE transactions on Software Engineering SE- 8(4): 437-439, July 1982

[26] www.gnu.org

[27] William Dickinson, David Leon and Andy Podgurski, "Pursuing Failure: The Distribution of Program Failures in a Profile Space", *Proc. 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering,* Vienna, Austria, *2001, pp.*: 246-255

[28] William Dickinson, David Leon, Andy Podgurski, "Finding failures by cluster analysis of execution profiles*", Proc. 23rd international conference on Software engineering*, Toronto, Ontario, Canada, 2001, pp.339-348.

[29] Michael R. Lyu, Zubin Huang, Sam K. S. Sze, Xia Cai, "An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering", *14th International Symposium on Software Reliability Engineering*, *Denver, Colorado*, November 17-21, 2003.

[30] David Leon, Andy Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases", *14th International Symposium on Software Reliability Engineering,* Denver, Colorado November 17-21, 2003.

[31] Norman E. Fenton and Niclas Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE transactions on Software Engineering*, Volume 26, No.8, August 2000, pp. 797-814.

[32] Thomas J. Ostrand, Elaine J. Weyuker, Robert M. Bell, "Where the bugs are", *Proc. of the 2004 ACM SIGSOFT international symposium on Software testing and analysis,* Boston, Massachusetts, USA, 2004, pp. 86 – 96.

[33] Thomas J.Ostrand, Elaine J. Weyuker, "The distribution of faults in a large industrial software system", *Proc. of the 2002 ACM SIGSOFT international symposium on Software testing and analysis,* Roma, Italy, 2002, pp. 55-64.

[34] Gregg Rothermel, Roland J. Untch, and Chengyun Chu, "Prioritizing Test Cases For Regression Testing", *IEEE Trans. Software Engineering*, Vol.27, No.10, 2001, pp.929-948.

[35] Jean Dolbec, and Terry Shepard "A component based software reliability model", *Proc. of the 1995 conference of the Center for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, 1995, pp.19-29.

[36] A.G. Koru and J. Tian, "Defect Handling in Medium and Large Open Source Projects", *IEEE Software*, Vol.21, No.4, pp.54-61, July/August, 2004.

[37] Robert M. Gray, “Entropy and Information Theory”, “Information systems laboratory electrical engineering department Stanford university”, Springer-Verlag, 19 November 2000.

[38] W. Farr, “Software Reliability Modeling Survey", *in Handbook of Software Reliability Engineering*, M.R. Lyu(Ed.) , McGraw-Hill, 1996, pp.71-117.

[39] G. Booch, J. Runbaugh, and I. Jacobson,, “*The Unified Modeling Language User Guide”*, Addison Wesley, 1998.

[40] http://xsuds.agreenhouse.com