Large Empirical Case Study of Architecture–based Software Reliability

Katerina Goševa-Popstojanova, Margaret Hamill, and Ranganath Perugupalli
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506-6109
{katerina, maggieh, perugupa}@csee.wvu.edu

Abstract

In this paper we present an empirical study of architecture–based software reliability based on a large open source application which consists of 350,000 lines of C code. The goals of our study are to analyze empirically the adequacy, applicability, and accuracy of architecture–based software reliability models. For this purpose we developed innovative approaches to efficiently extract and more accurately analyze a large amount of empirical data. Applying the theoretical results on a large scale field study allows us to test how and when they work, to understand their limitations, and outline the issues that need attention in the future research studies. Thus, our results show that for a subset of failures which can clearly be attributed to single components, both the composite and hierarchical models are very accurate when compared to the actual reliability. However, the assumptions made by the existing architecture–based software reliability models do not allow to account for the remaining failures which led to fixing faults in multiple components. These results show that in order to progress further, software reliability engineering should go through cycles of building models, testing them empirically, learning from the experiments, and refining the models to capture the newly discovered phenomena.

1 Introduction

Architecture–based assessment of software reliability combines the dynamic information about software architecture with the failure behavior of components. The motivation for the use of architecture–based software reliability approach includes the following:

- understanding how the system reliability depends on its components reliabilities and their interactions
- guiding the process of identifying critical components for a given architecture
- studying the sensitivity of the application reliability to reliabilities of components and interfaces
- selecting an architecture that is most appropriate for the system under study.

A number of models for estimating software reliability have been proposed in the past. However, there are many open questions with respect to the realism of the underlying assumptions, adequacy, accuracy, and applicability of these models. This paper is focused on empirical evaluation of these questions based on a real, large scale software application. We chose to use an open source software application as a case study since for a wide variety of open source projects many software artifacts necessary for architecture–based software reliability assessment are available. These include the source code, test suites (sometimes including test drivers and oracles), some type of change logs which keep track of changes made to the code and possibly other artifacts, and multiple releases of the software maintained using Concurrent Version System (CVS) [30].

Overall, open source software provides a unique opportunity for experimenting with large, realistic applications; it allows researchers to test the existing theoretical results and explore new domains, which over time will enable evolution of the knowledge and problem solving ability in software reliability engineering. We believe that, similarly to more mature fields such as physics and medicine, software reliability engineering will further advance because of the interaction between theoretical and experimental results. In these fields theorists build models that predict results of events that can be measured. These models may be based on theory or data from prior experiments. Then, experiments are carried out to test or disprove a theory, or explore a new domain. Regardless of the point the cycle is entered, there is a modeling, experimenting, learning, and remodeling pattern [1].

The rest of this paper is organized as follows. The related work and our contributions are discussed in section 2. The description of the case study and the experimental setup are presented in section 3. Detailed empirical results are given in section 4. In section 5 these results are used to estimate software reliability using composite and hierarchical architecture–based models. Finally, lessons learned and concluding remarks are given in section 6.

2 Related work and our contributions

Although numerous papers were devoted to architecture-based software reliability modelling (see for example [9] and references therein), most of them either do not include numerical illustrations [17], [20], [28] or illustrate the proposed models on simple made-up examples [2], [5], [15], [18], [29], [31].

Only a few papers so far applied the theoretical results on real case studies [8], [10], [11], [12], [14]. The analysis of a subsystem of a telephone switching system based on data collected from more than one thousand identical systems in several sites over three years was presented in [14]. The main sources of data were Failure Reports (FR) which described accurately the failure occurrence conditions and the consequences on the delivered

Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)
1071-9458/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

service, and Correction Reports (CR) which stated the technical reasons for failures, components concerned, and the corrections performed. The system was decomposed into four components accordingly to the four main functions. For each of these components the failure rate was estimated based on data extracted from 58 FR and 136 CR. However, the software execution behavior was not considered and the model of software architecture was not built.

In [8] the reliability of the SHARPE application was determined experimentally using the regression test suite which consisted of 735 test cases. SHARPE has over 35,000 lines of C code, 30 files, and a total of 373 functions. In this study each file was regarded as a single component. The software architecture represented by a discrete time Markov chain was determined using the coverage testing tool ATAC [32]. The execution counts and transition probabilities at the file level were obtained using the execution counts at the block level extracted from ATAC trace files. In this study, none of the test cases failed, so there was no need to detect any faults.

The case study from the European Space Agency which consists of almost 10,000 lines of C code was used in several of our earlier papers [10], [11], [12]. The application was divided into three components and the architecture was built using the component traces obtained during testing. The real faults detected during testing and operational usage were injected into the software, and the failures that happened as result of these faults were used to estimate components' reliabilities. Since the injected faults were known in advance, there was no need to identify faults in this study. The empirical data was used in [10] to estimate the point estimates of the reliability using several architecture–based models, and in [11] and [12] to study the uncertainty of reliability estimates.

A number of papers that presented empirical studies based on large software applications are related to our work, although their focus and goals were different. One group of papers presented empirical studies of large industrial software applications focused on identification of faults and prediction of fault–prone files [23], [24]. The empirical study of an inventory control system which had approximately 500,000 lines of code was presented in [23]. The authors proposed a statistical model to predict files that are most likely to contain a large number of faults during the system's next release. In [24] this model was used to predict fault–proneness on a file level for 17 releases of the inventory control system and 9 releases of the service provisioning system. These two studies were focused on prediction of fault–prone files based on the information about faults extracted from modification requests (i.e., change requests). One of the major problems was the identification of the modifications aimed at fixing faults, since modifications were also made to add new functionality or enhancements. Overall, in [23] and [24], the faults were identified on the file level, without collecting any information on software executions and considering the software architecture.

The difficulties encountered in conducting empirical studies on large industrial software systems were recently discussed in [25]. Accordingly to this paper the following issues led to difficulties while conducting the empirical studies presented in [23], [24]: extracting and analyzing large amounts of data, extracting from the repository which was not intended for the purpose of identifying faults, and difficulties in determining which modification requests represent faults.

Another set of empirical studies was focused on analysis of software execution profiles and using the results to draw conclusions about software failures and reliability. In [26], the authors applied clustering algorithms to partition execution profiles obtained during operational testing. Then, they used stratified random sampling to select executions for estimating the proportion of failures in the entire population of captured executions. The case studies considered in this paper ranged from 1,624 to 17,000 lines of code. Although the execution profiles were collected, the authors used a black–box reliability estimation method, which does not account for software architecture and components' reliabilities.

The work presented in [3] used clustering of software execution profiles aimed at predicting failures. Case studies considered in this paper were based on several Java programs (word count program *wc*, directory listing program *ls*, regular expression parser *rex*, regular expression finder *egrap*, and Java pretty printer *JSFormat*) and the *C compiler* of the *GNU Compiler Collection (GCC)* version 2.95.2. The results showed that a significant number of failures were isolated in small clusters of executions. Similar approach was taken in [4]; execution profiles which consisted of function–call counts were partitioned using clustering and it was shown that failures often have unusual profiles that were reviled by cluster analysis. Two empirical case studies were used in this paper: a music typesetting program *LilyPond* written in C++ with over 48,000 lines of code and *GCC C language compiler* with an overall size of 330,000 lines of code.

In [27] the authors proposed an automated support for classifying software failures in order to prioritize them and diagnose their causes. The classification strategies based on supervised and unsupervised pattern classification and multivariate visualization were applied on execution profiles. Three compilers, *GCC compiler for C*, and *Jikes* and *javac* Java compilers were used as subject programs in this paper. The execution profiles used in this study consisted of function-call counts which indicate the number of executions of each function during a test case execution. Unlike other papers [3], [4], this study included identification of faults based on execution of the same test suite on later software versions.

The same three open source applications, *GCC*, *Jikes*, and *javac*, were used in [19] for empirical comparison of coverage–based and distribution–based techniques for filtering large test suites. For this purpose the authors experimented with profiles with different granularity (functions, basic blocks, and control flow edges between basic blocks) using a modified version of the basic block profiler *gcov* [37] which is distributed with GCC.

In this paper we present an empirical study of architecture–based software reliability which is based on a significantly larger application then earlier studies [8], [10], [11], [12]. Furthermore, we conduct more comprehensive analysis than any of the previous empirical studies [8], [10], [11], [12], [14]. Some of the previously published empirical studies which used comparably large software applications were limited in scope and had different goals. Thus, one group of papers was focused on clustering of software execution profiles and using the results for prediction of failures [3], [4], [19], [26], [27], while the other was fo-

IEEE COMPUTER SOCIETY

cused on identification of faults based on modification records and proposed a method for prediction of fault–prone files [23], [24]. Next, we briefly compare our work to these studies.

We determine the dynamic aspects of the software architecture based on execution profiles obtained during testing with a regression test suite. This task requires resolving several challenging problems such as decomposition of the system into components in absence of complete and updated documentation, aggregating a large number of execution profiles which contain a significant number of function calls, and identifying the end points of software executions which are not given in the execution profiles under consideration. Unlike the previous work that was focused on clustering individual execution profiles in their original form [3], [4], [19], [26], [27], we aggregates a large amount of data extracted from the individual execution profiles in order to build a control flow graph which reflects the dynamic aspects of the software architecture.

As in the previous task, determining failure behavior on component level requires extracting large amounts of data. Additional difficulties arise from the fact that the available repositories are not intended for identifying faults and it is not trivial to distinguish changes made to fix faults from other changes such as enhancements and adding new requirements. Earlier work aimed at fault analysis for large case studies [23], [24] was focused only on identification of faults from modification (i.e., change) logs based on simplified heuristics. In our case, this information is not sufficient; we need to correlate failures of the software system with the corresponding faults that caused these failures and find faults' locations. Therefore, our approach includes software executions to detect failures, and then identification of faults that caused these failures based on more accurate methods which use other sources of information in addition to change logs.

Finally, we use both composite and hierarchical architecture–based models to combine the dynamic software architecture with components failure behavior. Our results show that both types of models give very accurate results when compared to the actual reliability. These results, however, are based on a subset of failures for which we are able to clearly identify components that caused these failures. Lessons learned from this work contribute towards enriching the empirical knowledge in software reliability and help in identifying important topics, such as the relationship between faults and failures, which require further research.

Accordingly to [1], the main signs of maturing in software engineering experimentation are: the level of sophistication of the goals of an experiment and its relevance to understanding interesting (e.g., practical) things about the field, and observing a pattern of knowledge building from a series of experiments. We believe that the research work presented in this paper contributes towards maturity of software reliability engineering experimentation.

3 Description of the case study and experimental setup

The GNU Compiler Collection (GCC) is an open source software which integrates compilers for several major programming languages including C, C++, Objective-C, Java, Fortran, Pascal, Mercury, Cobol, and Ada. For the experiments in this paper we use the *GCC C compiler* which has over 350,000 lines of code. GCC is a suitable case study for our experiments due to several reasons. It is a large scale application which allows us to study the applicability, accuracy, and scalability of architecture–based software reliability models. Like many open source projects, GCC uses a Concurrent Version System (CVS) as a version control tool. CVS stores the latest version of the code base, as well as the history of the code that was changed. Furthermore, a regression test suite is available with each version of GCC. This test suite is maintained by the GCC development team and it is open to the public for submission of additional test cases. When an unexpected output is found after a release of a GCC version, developers and users attempt to locate the fault and fix it. Changes made to the source code (e.g., fixing faults, enhancements, additions of new code) are recorded in *source code change log files*. Developers and users are encouraged to design and include new test cases in the regression test suite whenever changes are made to the source code. New test cases are recorded in the *test change log files* and become available with the next released version as a part of the regression test suite.

The test suite for GCC C compiler comes with test programs, drivers for these programs, and checkers that compare the execution behavior of compiled code to expected results. The drivers and the checkers play the role of a test oracle in this study. In our experimental setup we use the regression test cases of a newer version (i.e., GCC 3.3.3 released February 2004) to test an older version (i.e., GCC 3.2.3 released April 2003) since the test suites of the newer versions include tests for some faults present in the older versions. This is due to the fact that developers are encouraged to add test cases to the regression test suite whenever changes are made (including fixing faults). Thus, some test cases that do not fail when executed on the newer version will fail when executed on the older version. This process enables more failures to happen and allows us to detect higher number of faults. Similar approach was used in other empirical case studies, such as for example [3], [27], although their goal was different from ours.

It should be noted that the regression test suite represents one possible operational profile, which is not necessarily representative of the real usage of the GCC C compiler. This fact does not limit the validity of our results since our goal is to test empirically the theory of architecture–based software reliability rather than to estimate the reliability of GCC C compiler as seen by its users.

3.1 Our approach for empirical analysis of GCC

The common framework within which the existing architecture–based software reliability models are developed consists of the following steps [9], [10]: decomposition of the system to components, determination of software architecture, description of components' failure behavior, and combining the software architecture and failure behavior using analytical or simulation models.

Our approach for empirical analysis of GCC, presented in Figure 1, is a refinement of the informed approach introduced in [11] which is used during late phases of software development when testing or field data is available. Since the architecture–based approach requires insights into the software executions the first step is to instrument the software with a profiler [36], [37], or test coverage tool [32]. The left branch in Figure 1 represents the

IEEE COMPUTER SOCIETY

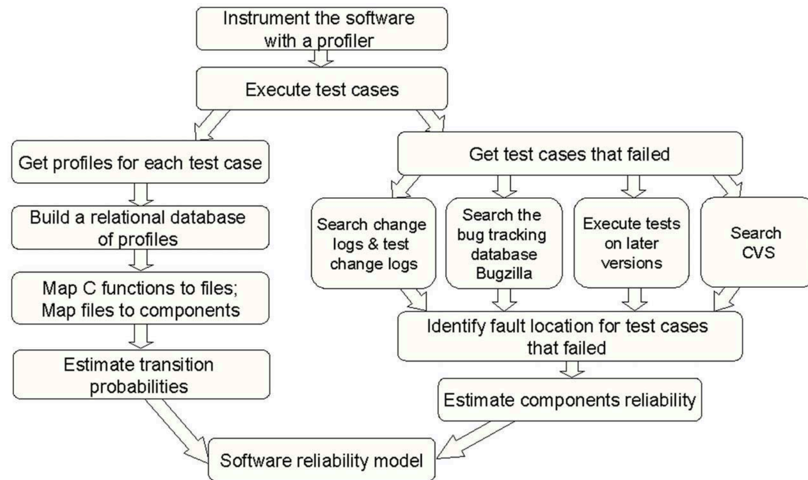

Figure 1. Our approach for empirical analysis of GCC

details of determining the software architecture, while the right branch represents the steps taken to determine components' failure behavior and estimate components' reliabilities. Finally, the dynamic information about the software architecture which includes the frequency of control transfer between components and the failure behavior of components described by components' reliabilities are combined using an architecture–based software reliability model.

In this paper we use state–based models to estimate software reliability [9], [10]. These models use the control flow graph to represent software architecture and assume that the transfer of control between components has a Markov property. In particular, we use models that represent software architecture with an absorbing Discrete Time Markov Chain (DTMC) with a transition probability matrix $P = [p_{ij}]$, where $p_{ij} = Pr\{\text{program transits from component } i \text{ to component } j\}$. Further, it is assumed that components fail independently and that a component failure will ultimately lead to a system failure. The reliability of the component i is defined as the probability R_i that the component performs its function correctly. The relevant measure for these models is the reliability R of a single execution of software application, that is, the so called probability of failure per demand $1 - R$. Next we present the detailed empirical results of our case study.

4 Empirical results

4.1 Determining the dynamic aspects of the software architecture

Software architecture reflects the way software components interact during execution. This means that we consider the dynamic information on interactions between components as a part of software architecture. As discussed in our earlier work [11] the dynamic software architecture can be determined in the early phases of the software life cycle using intended approach which may be based on specification, design, and documentation artifacts. In this paper we use informed approach which is based on information collected during software executions (e.g., integration testing or operational usage after deployment).

In our experimental study, we instrument the software with *gprof* [36] profiler to collect the information on software executions. When instrumented software is executed, a profile that characterizes the software execution for a given test case is generated. Gprof provides two types of execution profiles at a function level: flat profile and call graph. The flat profile is a simple function-call profile which has an entry for each function and gives the number of times the function was executed. The call graph is a function caller/callee profile which gives for a pair of functions `f` and `g` the number of times `f` called `g` during a corresponding execution. Since our model requires the knowledge of frequencies of control transfer, we use the call graph profile.

Out of 21,000 test cases in the regression test suite of GCC 3.3.3, 2,126 test cases are designed to test the C proper part of GCC. We ran these 2,126 test cases using the instrumented version of GCC 3.2.3. The corresponding 2,126 execution profiles contained 1,759 unique functions. Building an architecture–based software reliability model at the function level is not suitable because it leads to a large state space which poses difficulties in measurements, parameterization, and of the model. Furthermore, if we consider functions as components, the estimation of components' reliability may be statistically inaccurate due to a small size of functions and small number of functions' failures. To avoid these difficulties, we first mapped functions to files, and then grouped files into components that have clearly defined functionality and interfaces. The process of mapping functions to files was straightforward. For this purpose, we used the open source tool *ctags* [35] developed by GNU, which is used to extract different tags in C programs.[1] Using ctags, we found that the 1,759 functions belong to 108 source files in GCC. Our next step was to define components that have a clearly defined functionality and assign files to components. Based on the information provided in [38] we decomposed the system into 13 components. However, assigning files to components appeared to be difficult and time consuming process due to the fact that the documentation available on the GCC official Web site and other resources is outdated. Thus, out of 108 files which contained functions listed in the ex-

[1] A tag can be anything from a simple variable to something more complex, like a structure.

IEEE COMPUTER SOCIETY

ecution profiles, only 65 were mentioned in the documentation. The remaining 43 files were assigned to components based on our analysis of the source code and some help from GCC developers.

The next problem that needed to be resolved was to determine all functions (and correspondingly components) where the execution could end. This information is missing from the execution profiles produced by gprof which contain only the summary data on software executions (i.e., number of times a function called its children and was called by its parents) rather than a complete trace with a clear indication of the functions where the execution has terminated. Therefore, we were again forced to go through a tedious process of analyzing the code and determining the functions where the execution may end.

4.1.1 Estimation of transition probabilities

The point estimate of transition probabilities is obtained as $p_{ij} = n_{ij}/n_i$, where n_{ij} is the number of times the control has transferred from component i to component j, while $n_i = \sum_j n_{ij}$. The transition probabilities that define the transfer of control between software components are estimated from the whole set of execution profiles for all 2,126 test cases. In our case we have 2,126 execution profiles, each with over 700 unique functions. For the software of this size, it is not efficient to extract the information about the number of times the control passed from one component to another directly from the execution profiles (i.e., call graphs) produced by gprof which are in a plain ASCII format. We first parsed the execution profiles and created a database table with the profile ID, name of the caller function, name of the called function and the number of times it was called. This table has 4,643,491 rows which is a clear indication of the large amounts of data that needed to be analyzed. We also created a database table which contains the mapping of C functions to files and components. Querying these tables we created a new table which contains the number of times component i called component j. The transition probability matrix $P = [p_{ij}]$ of the DTMC which describes GCC architecture is given in Figure 2. It should be noted that executions always start in component 13. The matrix P has one additional row and column for the *End* state which represents the end of the executions.

4.2 Software failure behavior

One of the advantages of using the GCC regression test suite is that the information about whether each test case has failed or passed is automatically provided. This means that the time consuming manual inspection of the outputs to detect failures is not needed. When we ran the test cases using the make–check command various *.log and *.sum files were created in the subdirectories of the test suite. The results in the *.log and *.sum files are associated with the status codes shown in Table 1.

Status code	Meaning
PASS	Test passed as expected
XPASS	Test unexpectedly passed
FAIL	Test unexpectedly failed
XFAIL	Test failed as expected
UNSUPPORTED	Test is not supported on this platform
ERRORS	Test suite detected an error
WARNING	Test suite detected a possible problem

Table 1. Status codes used in *.log and *.sum files

We considered both FAIL and XFAIL status codes as failures and disregarded status codes UNSUPPORTED, ERROR, and WARNING since they do not represent failures of test cases. Thus, UNSUPPORTED status code is used to indicate test cases that are not supported on a given platform, while ERROR and WARNING are generated due to errors and possible problems in running the test cases such as for example failure of the test driver. The C proper part of GCC was executed on a total of 2,126 test cases out of which 111 failures (5.22% of all executions) were revealed by *.log, i.e., *.sum files.

The built–in test suite oracle was used by other researchers in the past for detection of test case failures [3], [4], [19], [27]. Although helpful, in our case this information is not enough. We also need to detect the location of faults that have caused the failures in order to be able to estimate components' reliabilities. Detection of faults that led to failures, especially for an application of this size, is not a trivial task. The three main sources of information that we used to detect faults are *CVS*, *test case change log files*, and *source code change log files*. Before we proceed with the methods that we used to identify faults that led to failures, we present a description of the GCC test case change log files and source code change log files.

To track the changes made to the test suite, developers maintain test case change logs. The first line in each entry consists of the date on which the test case has been written, the name of the author, and his or her e–mail address. The following lines list the test cases that were added by this author on this date. In some cases the Problem Report (PR) number is associated with the test case. This is important information that can be used to track faults. Surprisingly, out of 3,558 entries in the test case change logs only 697 had PR numbers. Out of these 697 PR numbers only 169 were tied to test cases used to test the C proper part of GCC.

Source code change log files are used to keep track of changes made to GCC source files; they are released with every version of GCC. The first line in each entry has the same format as in test case change logs – it contains the date when the change was made, and the name and e-mail address of the person that made the change. This line is followed by a list of files that were changed and a very brief description of the changes made. There were approximately 19,300 entries in source code change logs, out of which only 3,550 had PR numbers. As in the case of industrial case studies presented in [23], [24], there was generally no identification in the source code change logs whether a change was initiated because of fixing a fault, an enhancement, or some other reason such as change in the specification. In [23], [24] the authors used a heuristic to identify faults from modification request files which are similar to source code change log files in GCC. A rule of thumb used in these studies was that if only one or two files were changed by the modification request, then it was likely a fault, while if more than two files were affected, it was likely not a fault. Instead of using this kind of simplifying heuristics, we decided to develop more accurate methods for identification of faults. Even more, in our case it does not suffice to identify changes made to fix faults; we need to establish a cause–effect re-

IEEE COMPUTER SOCIETY

$$
\begin{bmatrix}
0 & 0.00053 & 0.13470 & 0 & 0 & 0 & 0.00003 & 0.06531 & 0.00471 & 0 & 0.18134 & 0.60951 & 0.00387 & 0 \\
0.39194 & 0 & 0.09140 & 0.01114 & 0 & 0 & 0 & 0 & 0.05387 & 0 & 0 & 0.45166 & 0.00000 & 0 \\
0.35396 & 0.00069 & 0 & 0.02593 & 0.00395 & 0.00559 & 0.00001 & 0.00468 & 0.02698 & 0 & 0.01639 & 0.56123 & 0.00060 & 0 \\
0.06769 & 0.20704 & 0.63969 & 0 & 0 & 0 & 0.04927 & 0.00045 & 0 & 0 & 0.03586 & 0 & 0 & 0 \\
0.38235 & 0 & 0.08589 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.38235 & 0.14941 & 0 \\
0.55386 & 0.04688 & 0.17313 & 0.00001 & 0 & 0 & 0 & 0.03471 & 0 & 0 & 0 & 0.19143 & 0 & 0 \\
0.03582 & 0.59501 & 0.16840 & 0.18777 & 0.00129 & 0 & 0 & 0 & 0 & 0 & 0.01171 & 0 & 0 & 0 \\
0.00680 & 0.02673 & 0.96213 & 0.00112 & 0 & 0 & 0 & 0 & 0.00093 & 0 & 0.00185 & 0.00044 & 0 & 0 \\
0.05920 & 0.00225 & 0.20549 & 0.00650 & 0 & 0 & 0 & 0.00520 & 0 & 0 & 0.00614 & 0.69656 & 0.01866 & 0 \\
0.06525 & 0.01021 & 0.75907 & 0.00520 & 0 & 0 & 0.14369 & 0.00681 & 0.00349 & 0 & 0.00340 & 0.00288 & 0 & 0 \\
0.05807 & 0.00088 & 0.05680 & 0 & 0 & 0 & 0 & 0 & 0.00000 & 0 & 0 & 0.88372 & 0.00053 & 0 \\
0.09972 & 0.00013 & 0.02055 & 0.00020 & 0 & 0 & 0 & 0 & 0.85341 & 0.00002 & 0.02595 & 0 & 0.00001 & 0 \\
0.07064 & 0.00572 & 0.08478 & 0.02827 & 0 & 0.00249 & 0.00613 & 0.01332 & 0.02698 & 0.00497 & 0.61022 & 0.04247 & 0 & 0.10402 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

Figure 2. Transition probability matrix P

lationship between faults and failures. The methods that we used to identify faults that caused 111 test cases to fail are described next.

4.2.1 Methods for fault identification

For our study, it would be ideal to have a PR number and the name of the test case that led to the failure associated with each entry in the source code change logs. This would allow us to distinguish all changes made to fix faults from other changes, as well as to associate them uniquely with the corresponding failures. Unfortunately, the names of test cases are rarely given in source code change log files and furthermore PR numbers are not always included in the entries for the changes made to fix faults. Therefore, we needed to develop methods that will allow efficient and accurate identification of faults that correspond to failed test cases. These methods, two automatic and two manual, are described next.

Method I: Searching test case change logs and source code change logs. From *.log and *.sum files we found out the names of all test cases that had failed. For each test case that failed, we searched the test case change logs to find the date and the author that has added this test case to the regression test suite. Using this information we searched the source code change logs. If the corresponding entry was found, it revealed the files that were changed to fix the fault(s) that caused the failure. In summary, this method ties the names of the failed test cases (indication of a failure) with the changes to files given in the source code change logs (indication of fixing faults) indirectly through information extracted from test case change log files. This method has several benefits. First, it is easy to automate, which is far more efficient and less fault–prone than a tedious manual search of the log files. (We used the Unix scripting language awk to write scripts that parse and search test case change logs and source code change logs.) More importantly, it associates failures with corresponding faults that caused these failures, i.e., reveals the cause–effect relationship between faults and failures. This is important since for our study it is not sufficient to identify faults. Last, but not least, this method allows us to distinguish between changes made due to fixing faults and changes made due to other reasons (e.g., enhancements or new requirements) without using heuristics based on rules of thumb such as for example in [23], [24]. This method was reasonably successful in identifying faults. Thus, we were able to locate faults that led to 43 out of 111 failures.

Method II: Searching the bug tracking database Bugzilla. Bugzilla is a free bug–tracking system which allows individuals or groups of developers to keep track of bugs in their product [33]. Our second method is based on searching Bugzilla for the Problem Report (PR) numbers given in GCC log files. First, we wrote awk script to search the test case change logs for PR numbers of failed test cases. Out of 111 failed test cases, only 24 had PR numbers. Then, we searched Bugzilla manually for these 24 PR numbers. Surprisingly, this method was not very successful in identifying faults; we identified the corresponding faults for only 6 failed test cases. The main reason for this poor result is that Bugzilla in most cases does not include the PR numbers.

Method III: Executing test cases on later versions and searching logs. Since we were not able to detect all faults leading to failures in the version under consideration (i.e., GCC 3.2.3), we executed the same test suite on later versions of GCC (i.e., GCC 3.3, 3.3.1, 3.3.2, and 3.3.3) in order to determine when the corresponding test case stopped failing. Similar method was used in the empirical studies presented in [19], [27]. After finding the version in which the fault was fixed, we repeated the first method to trace the location of the faults in files. For example, if test case *A* failed on version 3.2.3 and kept failing until version 3.3.2, it means that the corresponding fault(s) was (were) fixed in version 3.3.2. Therefore, we searched the source code change logs of the version 3.3.2 to localize the fault(s). Unlike Method I which requires only searching of test case change logs and source code change logs, Method III is more time consuming since it requires checking out, building, testing, and searching logs of multiple versions of GCC. However, it is still less time consuming and requires less domain and application knowledge than debugging the large application such as GCC. Using Method III, we were able to identify the faults that led to 24 failures.

Method IV: Searching CVS logs. For all the remaining failed test cases we searched the CVS logs available on the GCC Web site [34]. This method allowed us to find fixes that occurred days or even months after the test case was created. In many cases the comments in the CVS logs are more explanatory then those in the test case change logs and source code change logs. However, searching CVS manually and reading through logs to ensure which changes are responsible for fixing the failed test cases is time consuming due to the huge amount of data. Using this method we were able identify the changes made to fix the faults that led to failures of 12 additional test cases.

IEEE COMPUTER SOCIETY

Total number of failures	Method I	Method II	Method III	Method IV	Features not in GCC 3.2.3	Unresolved failures
111	43	6	24	12	7	19

Table 2. Distribution of the number of failures

Overall, we have identified the faults that led to 85 failures of GCC C compiler version 3.2.3. Since our experiments are based on testing the older version (i.e., 3.2.3) with the regression test suite of a newer version (i.e., 3.3.3), out of 111 failed test cases we have excluded 7 test cases which were designed to test features added to GCC C compiler after the version 3.2.3. This means that we were able to resolve 92 out of 111 failed test cases. The results of the process of identification of faults that led to failures are summarized in Table 2.

It is important to note that the four proposed methods allow us to identify only faults that have been already detected and fixed. Some of the 19 unresolved failures are due to faults that are either not known or not fixed yet and cannot be identified using any of the methods discussed here. Another reason for not being able to identify faults that led to some of the failures is the lack of consistency (or discipline) in the process of recording the fixes. This observation is consistent with the results from the survey of several open source projects presented in [13]. Out of 119 individual responses to the survey only 11.77% claimed that the defect tracking system was very consistent, that is, no defect gets fixed without reporting. 45.22% of the respondents answered that the defect tracking system is almost consistent, while the remaining 37.81% that it is not very consistent or not consistent.

4.2.2 Analysis of failure behavior

Once the faults that led to failures were identified, we associated these faults to the components identified in section 4.1 in order to be able to estimate components' reliabilities. A summary of files and components affected by fixing faults for 85 failures is given in Table 3[2]. Thus, 67.06% of failures were due to faults in one component, 21.18% to faults in two components, and 11.76% to faults in three to eight components. Similar results were obtained in [22] for a large industrial software application which consisted of 750,000 lines of code. In that study 15% - 23% of failures were associated with changing more than one component. Similarly, analysis of nearly two hundred anomalies from seven NASA spacecraft systems led to conclusion that some anomalies have multiple targets, that is, multiple corrections are made to fix the problem [21].

These observations clearly demonstrate that the relationship between faults and failures is complex and raise interesting questions, mainly unexplored in the literature. Since establishing links between faults and failures is not a trivial task, several simplistic assumptions were made in the past:

- In [21], for practical reasons, the first fix was selected to be recorded whenever multiple corrections were made to fix the problem. Although this solution is practical, it masks the remaining fixes.

- In [22] a different approach was taken; if a cause of a failure was corrected by modifying n files (i.e., multiple targets) n distinct faults were counted which assures that every fault is associated with a unique file. However, this study was focused only on faults, that is, it did not attempt to analyze the links between faults and failures.

- In [24] and [25], in order to simplify the identification of faults from modification records, it was assumed that only changes made to one or two files are related to fixing faults. Our results show that 30.59% of failures required fixing more than 2 files (see Table 3). Obviously, using heuristics as this one is not justified and may lead to significant errors in the analysis.

- In [7] it was assumed that all failures are traced back to a unique fault in a module. A similar assumption is made in most software reliability growth models [6]. Although this assumption simplifies the analysis, obviously, it is not realistic. In our case, 32.94% of failures were tracked to faults in more than one component.

- In [27], for simplicity it was assumed that each failure is caused just by one fault. Based on this assumption the set of failures is partitioned in into k subsets such that all of the failures in a given subset is are caused by the same fault. Our results show that this assumption is not valid. Thus, in some cases even when a failure can be associated with a single component it is due to multiple faults.

Number of files affected	Failures	% of failures	Number of components affected	Failures	% of failures
1	36	42.35	1	57	67.06
2	23	27.06	2	18	21.18
3	6	7.06	3	5	5.88
4	12	14.12	4	3	3.53
5	2	2.35	5	1	1.18
6	2	2.35	8	1	1.18
8	3	3.53			
14	1	1.18			

Table 3. Summary of faults distribution across files and components

The existing architecture–based software reliability models assume that components fail independently and a component failure leads to a system failure. As the results in Table 3 show, we can be confident that this assumption is valid for 67.06% of the failures. For the remaining 28 failures (i.e., 32.94%) the current state of practice for tracking problem reports and code changes does not allow the process to be fully automated. Instead, we examined manually the records in the CVS and source code change logs which corresponded to fixing multiple components related to the same failed test case. For 12 of them we were able to assign

[2]Components consist of 1–32 files.

IEEE COMPUTER SOCIETY

the cause of the failure to a single component. Due to the lack of information in source code change log files and CVS, the large scale of the application, and lack of domain knowledge, we were not able to draw any conclusions about 16 failed test cases which led to changes in multiple components. For example, the person who made changes to several components tied to the same failed test case wrote a comment "Initialize variable" for the first file changed, and then added "Likewise" for several other files. From this record in source code change log file it cannot be concluded whether the failure resulted as a combined effect of these faults or the developer has made similar changes to multiple files, not all of them necessarily related to this particular failure.

These results show that the relationship between faults and failures is not trivial. In addition to the intrinsic complexity of the problem, the current defect tracking systems are not intended to be used for this type of analysis and very often contain incomplete and inconsistent data. It is obvious that a series of case studies and controlled experiments are needed to build a pattern of knowledge in this area.

4.2.3 Estimation of components' reliabilities

Further study of the fault–failure relationship is out of the scope of this paper. Instead of making simplifying assumptions, we decided to consider only the 57 failures that were caused by faults in a single component for estimation of components' reliabilities. Thus, we consider a subset of the regression test suite as an operational profile for the system. Our future work will address the problem of how to account for failures that require changes in multiple components.

The point estimate of the reliability of component i is obtained as $R_i = 1 - f_i/n_i$, where f_i is the number of failures in n_i executions of component i. Note that the number of component executions n_i is a random variable whose value was estimated querying the database of execution profiles. The values of components' reliabilities are given in Table 4.

Component	R_i	Component	R_i
1	0.99999929	8	1
2	0.99999946	9	1
3	0.99999974	10	1
4	1	11	0.99999972
5	0.99999342	12	0.99999989
6	0.99999539	13	0.99999872
7	1		

Table 4. Point estimates of components' reliabilities

5 Empirical assessment of architecture–based reliability

Accordingly to the solution method, state–based software reliability models can be classified as either composite or hierarchical [9], [10]. The composite method combines the architecture of the software with the failure behavior into a composite model which is then solved to predict the reliability of the application. The other possibility is to take a hierarchical approach, that is, to solve first the architectural model and then to superimpose the failure behavior on the solution of the architectural model in order to predict reliability. Next, we briefly describe the composite and hierarchical models used for reliability assessment in this study.

The model presented in [2] uses a composite solution method; two absorbing states C and F, representing the correct output and failure respectively, are added to the DTMC. The transition probability matrix P is modified to $\hat{P}$ as follows. The original transition probability p_{ij} between the components i and j is modified into $R_i\, p_{ij}$, which represents the probability that the component i produces the correct result and the control is transferred to component j. From the exit state n, a directed edge to state C is created with transition probability R_n to represent the correct execution. The failure of a component i is considered by creating a directed edge to failure state F with transition probability $(1 - R_i)$. The reliability of the program is the probability of reaching the absorbing state C of the DTMC. Let Q be the matrix obtained from $\hat{P}$ by deleting rows and columns corresponding to the absorbing states C and F. The $(1, n)$ element of matrix Q^k represents the probability of reaching state n from 1 through k transitions. From initial state 1 to final state n, the number of transitions k may vary from 0 to infinity. It can be shown that $S = \sum_{k=0}^{\infty} Q^k = (I - Q)^{-1}$, so it follows that the overall system reliability can be computed as

$$R = s_{1,n}\, R_n, \tag{1}$$

where $s_{1,n}$ is the $(1, n)$ element of matrix S. Basically, the reliability is equivalent to the sum of reliabilities of all paths that start at the entry node and end at the exit node C, including the possibility of an infinite number of paths due to the loops that might exist between two or more components.

For the hierarchical solution approach we use the model presented in [10], which is a variant of the models proposed in [8], [16]. This model establishes a link between hierarchical models, while using the same input data as the composite model [2] which allows us to compare the results. The solution of the hierarchical models is based on the following reasoning. During a single execution of the software application each component i is executed a random number of times, denoted by N_i. Thus $R_i^{N_i}$ can be considered as the equivalent reliability of component i that takes into account its utilization. Assuming that components fail independently, the system reliability becomes $\prod_{i=1}^{n} R_i^{N_i}$. The first order approximation of Taylor's series expansion of $E[\prod_{i=1}^{n} R_i^{N_i}]$ is then used as an approximate estimate of the system reliability

$$R \approx \prod_{i=1}^{n} R_i^{V_i} \tag{2}$$

where $V_i = E[N_i]$ is the expected number of times component i is executed during a single execution of a software. This approximation is based on the assumption that components are highly reliable and variances of the number of times each component is executed are very small. V_i are obtained by solving the following system of linear equations

$$V_i = q_i + \sum_{j=1}^{n} V_j\, p_{ji} \tag{3}$$

IEEE COMPUTER SOCIETY

where q_i denotes the initial probability of state i, that is, the probability that the software execution starts from component i.

We estimate the actual reliability of the software as

$$R = 1 - \frac{F}{N} \quad (4)$$

where $F = 57$ is the number of system failures in $N = 2,072$ test cases[3]. As it can be seen from Table 5, the composite model (1) and hierarchical model (2) give extremely close results with high accuracy when compared to the actual reliability (4).

	Reliability	Error
Actual	0.9724903475	
Composite	0.9997856425	2.8067%
Hierarchical	0.9997928951	2.8075%

Table 5. Comparison of the results

It should be noted that the approximate solution of the hierarchical method might not be always so close to the exact solution of the composite method, as we have shown in our earlier work [10]. Furthermore, due to independence assumption the system reliability may be underestimated by either of these methods, especially if the components that are executed many times during a single application execution are less reliable.

6 Lessons learned and concluding remarks

In this paper we have presented an empirical study of architecture–based software reliability based on a real, large scale software application. Our results are based on innovative approaches to efficiently extract and more accurately analyze large amounts of empirical data needed for architecture–based software reliability assessment. To the best of our knowledge, this is the largest and the most comprehensive empirical study ever used in architecture–based software reliability.

Lessons learned from conducting the empirical study presented in this paper include:

- *Large quantity of data has to be extracted and analyzed.* In our empirical study we dealt with over 4.5 million entries in execution profiles used to build the software architecture and almost 23,000 entries in source code change logs and test case change logs used to identify faults that led to failures. Therefore, for large systems manual examination of the execution profiles and change logs is almost impossible. Rather, automatic methods for efficient data extraction and analysis are needed.

- *Decomposition of the system into components may not be an easy task.* Even when the decision about the components can be easily made based on well known functionalities of the system, conducting the decomposition may be difficult due to the large scale of the system and outdated documentation.

- *Identification of faults that led to failures is not trivial.* Although most systems keep track of changes in some form of change logs, the reasons why changes were made usually are not identified explicitly. Therefore, it is not trivial to distinguish changes made to fix faults from other changes made for example to add new functionality or planned enhancements. Better format for keeping track of problem reports and changes made to the source code needs to be developed and adopted in practice.

- *Relationships between faults and failures are complex and almost unexplored in the literature.* Our results show that many simplifying assumptions made in the past are not valid and may lead to errors in the analysis. More empirical and theoretical research in this area should be conducted by both software testing and reliability engineering research communities.

- *Both the composite and hierarchical models are very accurate when compared to the actual reliability.* These estimates, however, are based on a subset of failures which can clearly be attributed to single components. Once more sound relationships between faults and failures are established, the current state of the art in architecture–based reliability has to be enhanced to account for them.

The results from this research undoubtedly show that open source software holds enormous potential for enriching the empirical knowledge in software reliability. They also show that theoretical research results have to be applied on real, large scale field studies to examine how and when they really work, to understand their limits, and to understand how to improve them. In particular, the empirical results presented in this paper pinpoint some challenging problems which were not addressed in the previous work on software reliability. Additional empirical studies are needed to enhance the understanding of the complex and mainly unexplored relationships between faults and failures. Subsequently, theoretical research should follow to account for the newly discovered phenomena.

[3] The 7 test cases testing features not in GCC 3.2.3, 19 unresolved failures, and 28 failures that led to fixing faults in multiple components are excluded both from the failed and the total number of test cases.

Acknowledgements

This work is funded in part by grant from the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA IV&V Facility, Fairmont, West Virginia. The authors thank the contributors of GCC who helped clarifying the quality assurance practice used in the GCC project.

References

[1] V. R. Basili, "The Role of Experimentation in Software Engineering: Past, Current, and Future", *Proc. 18th International Conference on Software Engineering*, 1996, pp. 442–449.

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

[3] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles", *Proc. 23rd International Conference on Software Engineering*, 2001, pp. 339–348.

[4] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing Failure: The Distribution of Program Failures in a Profile Space", *Proc. 10th*

European Software Engineering Conference and 9th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2001, pp. 246–255.

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

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

[7] N. E. Fenton and N. Ohisson, "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Transactions on Software Engineering*, Vol.26, No. 8, August 2000, pp. 797–814.

[8] S. Gokhale, W. E. Wong, K. Trivedi, and J. R. Horgan, "An Analytical Approach to Architecture–Based Software Reliability Prediction", *Proc. 3rd International Computer Performance and Dependability Symposium*, 1998, pp. 13–22.

[9] K. Goševa–Popstojanova and K. Trivedi, "Architecture–Based Approach to Reliability Assessment of Software Systems", *Performance Evaluation*, Vol. 45, 2001, pp. 179–204.

[10] K. Goševa–Popstojanova, A. P. Mathur, and K. S. Trivedi, "Comparison of Architecture-Based Software Reliability Models", *Proc. 12th International Symposium on Software Reliability Engineering*, 2001, pp. 22–31.

[11] K. Goševa–Popstojanova and S. Kamavaram, "Assessing Uncertainty in Reliability of Component-Based Software Systems", *Proc. 14th IEEE International Symposium on Software Reliability*, 2003, pp. 307–320.

[12] K. Goševa–Popstojanova and S. Kamavaram, "Software Reliability Estimation under Uncertainty: Generalization of the Method of Moments", *Proc. 8th IEEE International Symposium on High Assurance Systems Engineering*, 2004, pp. 209–218.

[13] A. Gunes Koru and J. Tian, "Defect Handling in Medium and Large Open Source Projects", *IEEE Software*, July/August 2004, pp. 54–61.

[14] K. Kanoun and T. Sabourin, "Software Dependability of the Telephone Switching System", *Proc. 17th International Symposium on Fault Tolerant Computing*, 1987, pp. 236–241.

[15] S. Krishnamurthy and A. Mathur, "On the Estimation of Reliability of a Software System using Reliabilities of its Components", *Proc. 8th International Symposium on Software Reliability Engineering*, 1997, pp. 146–155.

[16] P. Kubat, "Assessing Reliability of Modular Software", *Operations Research Letters*, Vol. 8, 1989, pp.35-41.

[17] J-C. Laprie, "Dependability Evaluation of Software Systems in Operation", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, 1984, pp. 701–714.

[18] J. Ledoux, "Availability Modeling of Modular Software", *IEEE Transactions on Reliability*, Vol. 48, No. 2, 1999, pp. 159–168.

[19] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases", *Proc. 14th IEEE International Symposium on Software Reliability*, 2003, pp. 442–453.

[20] B. Littlewood, "Software Reliability Model for Modular Program Structure", *IEEE Transactions on Reliability*, Vol. R-28, No.3, 1979, pp. 241–246.

[21] R. R. Lutz and I. C. Mikulski, "Empirical Analysis of Safety Critical Anomalies During Operation", *IEEE Transactions of Software Engineering*, Vol. 30, No.3, March 2004, pp. 172–180.

[22] K. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution", *Proc. 1st IEEE International Software Metrics Symposium*, 1993, pp. 82–90.

[23] T. J. Ostrand and E. J. Weyuker, "The Distribution of Faults in a Large Industrial Software System", *Proc. ACM International Symposium on Software Testing and Analysis*, 2002, pp. 55–64.

[24] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are", *Proc. ACM International Symposium on Software Testing and Analysis*, 2004.

[25] T. J. Ostrand and E. J. Weyuker, "Difficulties Encountered Doing Empirical Studies in an Industrial Environment", *Supplementary Proc. 15th IEEE International Symposium on Software Reliability*, 2004, pp. 17–18.

[26] A. Podgurski, W. Masri, Y. McCleese, F. Wolff, and C. Yang, "Estimation of Software Reliability by Stratified Sampling", *ACM Transactions on Software Engineering and Methodology*, Vol. 8, No.3, July 1999, pp. 263–283.

[27] A. Podgurski, D. Leon, P. Franis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated Support for Classifying Software Failure Reports", *Proc. 25th International Conference on Software Engineering*, 2003, pp. 465–475.

[28] M. Shooman, "Structural Models for Software Reliability Prediction", *Proc. 2nd International Conference on Software Engineering*, 1976, pp. 268–280.

[29] H. Singh, V. Cortellessa, B. Cukic, E. Guntel, and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems", *Proc. 12th International Symposium on Software Reliability Engineering*, 2001, pp. 12–21.

[30] L. Zhao and S. Elbaum, "Quality Assurance under Open Source Development Model", *Journal of Systems and Software*, Vol.66, 2003, pp. 65–75.

[31] S. Yacoub, B, Cukic, and H. Ammar, "Scenario-Based Reliability Analysis of Component-based Software", *Proc. 10th International Symposium on Software Reliability Engineering*, 1999, pp. 22–31.

[32] http://xsuds.argreenhouse.com

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

[34] http://gcc.gnu.org/

[35] http://ctags.sourceforge.net/ctags.html

[36] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html

[37] http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/Gcov-Intro.html#Gcov-Intro

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

Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)
1071-9458/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY