Adequacy, Accuracy, Scalability, and Uncertainty of Architecture–based Software Reliability: Lessons Learned from Large Empirical Case Studies

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

Abstract

Our earlier research work on applying architecture–based software reliability models on a large scale case study allowed us to test how and when they work, to understand their limitations, and to outline the issues that need future research. In this paper we first present an additional case study which confirms our earlier findings. Then, we present uncertainty analysis of architecture–based software reliability for both case studies. The results show that Monte Carlo method scales better than the method of moments. The sensitivity analysis based on Monte Carlo method shows that (1) small number of parameters contribute to the most of the variation in system reliability and (2) given an operational profile, components' reliabilities have more significant impact on system reliability than transition probabilities. Finally, we summarize the lessons learned from conducting large scale empirical case studies for the purpose of architecture–based reliability assessment and uncertainty analysis.

1. Introduction

Although the architecture–based estimation of software reliability has a long tradition, there are still many open questions with respect to the realism of the underlying assumptions, adequacy, accuracy, and scalability of these models. The goal of this paper is to evaluate empirically these questions based on large scale software applications. It applies the experimental set-up and methods for data extraction presented in our earlier work [11] on additional case study which allows us to further explore the architecture–based software reliability models empirically and draw common conclusions. Furthermore, we apply the methods for uncertainty analysis developed in our earlier work [9], [10] on these two large scale case studies and study how the uncertainty of the parameters values propagates into the uncertainty of the reliability estimate.

The type of empirical studies presented in this paper is called observational since, unlike controlled experiments, the subjects under study are not perturbed. Observational studies are typically easier to plan and scale better than controlled experiments. Further advantages include the capability to incorporate complexity, unpredictability, and dynamism. The fact that observational studies that include several subjects can be considered as one form of replicated experiment provides basis for generalization of results. However, it is harder to interpret the results of the observational studies since researchers do not have the same control as in experiments.

Next, we summarize the related work and emphasize our contributions. Although numerous papers were devoted to architecture-based software reliability modelling, most of them either do not include numerical illustrations [14], [15] or illustrate the models on simple made-up examples [1], [13], [23], [24]. A few papers that so far applied the theoretical results on real case studies did not include building the software architecture [12] or identification of faults [6], [8], [9], [10].

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. Next, we briefly compare our work to these studies. Unlike the previous work that was focused on clustering individual execution profiles in their original form [2], [3], [21], we build the dynamic aspects of the software architecture based on the data extracted from the individual execution profiles. This task requires resolving several challenging problems such as decomposition of the system into components, aggregating a large number of execution profiles, and identifying the end points of software executions. Earlier work aimed at fault analysis for large case studies [19], [20] was only focused on identification of faults from modification (i.e., change) logs based on simplified heuristics. In our case, this information is not sufficient. Therefore, our approach includes software executions to detect failures, and then identification of the location of faults that caused these failures based on several accurate methods.

Last but not least, we build the architecture–based software reliability models and conduct uncertainty analysis based on Monte Carlo simulation and method of moments. It should be emphasized that we have generalized our earlier results on Monte Carlo simulation and method of moments [9], [10] to account for the uncertainty in (1) the reliability estimates for components that in their current version do not reveal any failures and (2) transition probability estimates for potential transitions that are not observed during specific number of executions.

The rest of the paper is organized as follows. The methods for uncertainty analysis are briefly described in section 2. The experimental setup and the empirical results from the two case

17th International Symposium on Software Reliability Engineering (ISSRE'06)
0-7695-2684-5/06 $20.00 © 2006 IEEE

studies are presented in section 3. The methods for uncertainty analysis are applied on both case studies in section 4. Finally, lessons learned and concluding remarks are given in section 5.

2. Methods for uncertainty analysis

In order to estimate the system reliability using architecture–based model we need to know the dynamic aspects of the software architecture (structure and relative frequencies of components interactions) and software failure behavior (components' reliabilities or failure rates). We use state–based approach to build the architecture–based software reliability model [7]. In this approach states represent active components and arcs represent the transfer of control. First, the architecture is modeled with a discrete time Markov chain (DTMC) with a transition probability matrix $P = [p_{ij}]$, where $p_{ij} =$ Pr {control is transferred from component *i* to component *j*}. Then, components failure behavior is considered (i.e., reliability R_i of each component is estimated) and a model that combines software architecture with components failure behavior is built. Our methodology for uncertainty analysis [9] is not tied to a particular architecture–based software reliability model. In this paper we use the model first presented in [1] which uses composite method to combine software architecture with failure behavior and provides close form solution for the software reliability as function of components' reliabilities and transition probabilities $R = f(R_i, p_{ij})$.

For a given software architecture, there are two sources of uncertainty in software reliability: the way components interact (i.e., transition probabilities) and the components' failure behavior (i.e., components' reliabilities). Regardless of the accuracy of the mathematical model, if considerable uncertainty exists in estimation of the parameters (as it usually does), the traditional approach of computing the point estimate of the software reliability is not appropriate. Alternatively, we can treat unknown parameters as random variables and quantify the uncertainty of system reliability. In this paper we use two methods for uncertainty analysis: Monte Carlo simulation [9] and method of moments [10].

2.1. Monte Carlo simulation

Monte Carlo simulation is an approximate method for estimating reliability of the system when the parameters can be represented by well defined probability distributions. The approach we used in our earlier work [9] did not allow us to take into account the uncertainty due to the following cases:

- *Observing zero failures for some components* $f_i = 0$. The traditional reliability estimate for a software component that in its current version has not failed will result in reliability equal to 1. Even worse, when none of components have failed (i.e., the testing does not reveal any failures), the estimate of the system reliability will be equal to 1. Of course, unless we do exhaustive testing without replacement, we can never be sure that the reliability of a software component or a software system is 1.
- *Observing zero counts for the transfer of control (i.e., transitions)* $n_{ij} = 0$. Not all transitions with zero counts are improbable. In some cases, zero count might mean that none of the test cases have triggered the transfer of control between components i and j. Thus, potential transitions should not be excluded simply because they were not observed during specific number of executions.

The classic solution to the problem of estimating components reliability based on failure–free executions is based on the Bayesian framework [17]. The number of successes r_i in n_i executions, given component reliability R_i ($0 \leq R_i \leq 1$), follows the binomial distribution

$$\left(\begin{array}{c} n_i \\ r_i \end{array} \right) R_i^{r_i}(1-R_i)^{n_i-r_i}. \tag{1}$$

Within the Bayesian framework a priori knowledge about the parameter of interest, here R_i, is represented by the prior distribution. In this case we use as a prior distribution the conjugate distribution Beta(a_i, b_i) given with equation

$$f(R_i) = \frac{\Gamma(a_i+b_i)}{\Gamma(a_i)\Gamma(b_i)} R_i^{a_i-1}(1-R_i)^{b_i-1} \tag{2}$$

where $a_i > 0$ and $b_i > 0$. Here we concentrate on the case when no prior information is available and use the "ignorance" uniform prior Beta$(1,1)$ in which case the posterior distribution reduces to Beta(r_i+1, n_i-r_i+1).

We solved the problem of observing zero counts for the transfer of control in two steps. First, we ran a static code analysis tool to determine whether the transition is not possible (i.e., the transition count will always be zero $n_{ij} = 0$) and therefore $p_{ij} = 0$. When the static code analysis shows that the transition is possible, but no transitions were observed during specific number of executions, the transition probability is likely to be close to 0, but is not improbable. To account for this case, as in the case of components reliability when no failures were observed, we use Bayesian framework.

For each component, the control will be transferred to one of at most n components. Let r_{ij} denote the number of times the control was passed from component i to component j in n_i executions. Then, the data follows the multinomial distribution

$$\left(\begin{array}{c} n_i \\ r_{i1} r_{i2} \ldots r_{in} \end{array} \right) p_{i1}^{r_{i1}} p_{i2}^{r_{i2}} \ldots p_{in}^{r_{in}} \tag{3}$$

for $r_{ij} = 0, 1, 2, \ldots, n_i$ and $\sum_{j=1}^{n} r_{ij} = n_i$. It is important to emphasize that the number of categories in the multinomial distribution will typically be less than n because, as described earlier, some of the transitions are improbable. We assume that the rows in the transition probability matrix are independent and distributed accordingly to Dirichlet distribution, that is, for the *i*th row in the transition probability matrix we choose Dirichlet prior distribution Dirichlet$(\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in})$ given by

$$f(p_{i1}, \ldots, p_{in}) = \frac{\Gamma(\alpha_{i1} + \ldots + \alpha_{in})}{\Gamma(\alpha_{i1}) \ldots \Gamma(\alpha_{in})} \prod_{j=1}^{n} p_{ij}^{\alpha_{ij}-1} \tag{4}$$

where $\alpha_{i1}, \ldots, \alpha_{in} > 0$, $p_{ij} \geq 0$, and $\sum_{j=1}^{n} p_{ij} = 1$. As in case of components reliabilities, we use the "ignorance" uniform prior Dirichlet$(1, 1, \ldots, 1)$, which leads to posterior distribution Dirichlet$(r_{i1}+1, r_{i2}+1, \ldots, r_{in}+1)$.

2.2. Method of moments

Method of moments is an approximate analytical approach that can be applied to any architecture-based software reliability model that has a close form solution for the system reliability. It consists of expanding $R = f(R_i, p_{ij})$ by a multivariable Taylor series about the point at which components reliabilities and transition probabilities take their expected values. Method of moments is an approximate method because of the omission of higher order terms in the Taylor series expansion. The expressions for the mean $E[R]$ and the variance Var$[R]$ of the system reliability based on the first and second order Taylor series expansion, which are not given here due to space limitation, can be found in [10]. Since deriving the expression for the system reliability and the corresponding Taylor coefficients by hand is cumbersome and can be done only for small systems, we used *Mathematica* to automate the process.

In general, method of moments only requires sample estimates of the moments of components' reliabilities and transition probabilities (i.e., no distribution must be specified). However, in cases when zero failures are observed for some components or zero counts are observed for some probable transitions, sample means and higher central moments will be zero and will not allow to account for uncertainty of these parameters. In these cases, it is necessary to use Bayesian approach and estimate the moments of the posterior distributions given in section 2.1.

3. Empirical results

In this section we briefly summarize the results for the GCC C compiler given in our earlier work [11] and present the results of a new case study Indent. We chose these two applications due to the following reasons. (1) Both projects use Concurrent Version System (CVS) and have multiple releases available. (2) A regression test suite is available with each release of the software, including drivers to automatically run the test suite and checkers that compare the outputs of the test cases to the expected results. The drivers and the checkers play the role of a test oracle in our studies. (3) Changes made to the source code (e.g., fixing faults, adding new functionality) are recorded in *source code change log files*. Developers of GCC also record new test cases in the *test change log files*. (4) Different sizes of the case studies allow us to explore the scalability of architecture–based software reliability models.

In our experimental setup we use the regression test cases of a newer version to test an older version, which enables more failures to happen and allows us to detect higher number of faults. Special care was taken to exclude test cases (including failures) designed to test features implemented in newer versions. The basic facts of the case studies are as follows.

- *GCC C compiler* is a part of the GNU Compiler Collection [26]. We experimented with 5 releases of GCC. The latest considered release, 3.3.3, has over 300,000 lines of code and 2,126 test cases designed to test the C proper part of GCC.
- *Indent* [27] is a C code beautifier which changes the appearance of C programs. We experimented with 10 releases of Indent. The latest considered release 2.2.9 has around 11,000 lines of code and regression test suite with 158 test cases.

It should be noted that the regression test suites represent one possible operational profile, which is not necessarily representative of the real usage of GCC C compiler and Indent. However, 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 as seen by the users.

3.1. Determining dynamic aspects of the software architecture

To collect the information on software executions we instrument the software applications with *gprof* profiler [28]. Since our model requires the knowledge of frequencies of control transfer, we use the call graph profiles produced by gprof.

We ran the 2,126 test cases designed to test the C proper part of GCC 3.3.3 using the instrumented version of GCC 3.2.3. The corresponding 2,126 execution profiles contained 1,759 unique functions. We mapped these 1,759 functions to 108 files, and then grouped these files into 13 components that have clearly defined functionality and interfaces. While mapping functions to files was straightforward, assigning files to components appeared to be difficult and time consuming process since the available documentation was outdated. The process of identifying all functions (and correspondingly components) where the execution could end was not trivial as well since this information is not contained in the execution profiles produced by gprof. Analyzing the code of GCC C compiler we concluded that each software execution started and ended in functions that belong to component 13.

We ran 158 test cases of Indent version 2.2.9 using the instrumented version of Indent 2.2.0. We determined that the 58 unique functions shown in the execution profiles belong to 9 files and we decided to build the architecture–based software reliability model at file level. Following the same process as in case of GCC, we determined that each software execution started and ended in component 6.

3.2. Determining software failure behavior

One of the advantages of using GCC and Indent regression test suites is that the information about whether each test case has failed or passed is automatically provided. Although helpful, in our case this information is not enough. In order to be able to estimate components' reliabilities, it is required to detect the location of faults that had caused the failures, that is, to establish a cause–effect relationship between faults and failures which for applications of this size is not a trivial task.

The C proper part of GCC was executed on a total of 2,126 test cases out of which 111 failed. Instead of using some kind of simplifying heuristics, we decided to develop accurate methods for identification of the faults that led to these failures. These methods, two automatic and two manual, are briefly described next. For detailed description the reader is referred to [11].

Method I 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. Method II is based on searching the bug–tracking system Bugzilla [25] for the Problem Report numbers given in GCC log files. Method III consists of executing 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. 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 all the remaining failed test cases we searched manually the CVS logs available on the GCC Web site [26].

Overall, we identified the faults that led to 85 failures of GCC C compiler version 3.2.3 (43, 6, 24 and 12 using Methods I-IV, respectively). In addition, we identified that seven test cases failed because they were designed to test features added after the version 3.2.3. This means that we were able to resolve 92 out of 111 failed test cases. 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 above. 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.

A summary of files and components affected by fixing faults for 85 failures is given in Table 1. 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 [18] 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 200 anomalies from seven NASA spacecraft systems led to conclusion that multiple corrections are made to fix some anomalies [16].

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 1. Faults distribution for GCC

Out of 158 test cases available in the regression test suite of Indent version 2.2.9, 34 test cases failed when executed on Indent version 2.2.0. For identification of the location of faults that led to these 34 failures, we used Method III described earlier in this section. The .diff files which showed the difference between the expected and the actual output file were also helpful. Overall, we identified faults that led to 27 failures. Three additional failures were excluded since they were designed to test features added to Indent after the version 2.2.0. We could not identify the faults that led to four failures. One of these four failed test cases was still failing on version 2.2.9, which means that the faults related to this failure were not fixed.

As it can be seen from Table 2, 88.89% of Indent failures were due to faults in one file, that is, 11.11% of failures were caused by faults is two files. Obviously, the fault–failure relationships for smaller case studies such as Indent tend to be simpler when compared to larger and more complex case studies such as GCC C compiler.

Number of files affected	Failures	% of failures
1	24	88.89
2	3	11.11

Table 2. Faults distribution for Indent

The results of our experiments 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. For example, in [5] and [21] it was assumed that failures are traced back to a unique fault in a module. A similar assumption is made in most software reliability growth models [4]. Although this assumption simplifies the analysis, obviously, it is not realistic. 32.94% GCC C compiler failures and 11.11% of Indent failures were tracked to faults in more than one component. Another example is the heuristic used in [19] and [20] which was based on the assumption that only changes made to one or two files are related to fixing faults. Although the results for Indent, which is considerably smaller and simpler application, agree with this heuristic, the results for GCC C compiler show that 30.59% of failures required fixing faults in more than 2 files (see Table 1).

The existing architecture–based software reliability models assume that components fail independently and each component failure leads to a system failure. As the results in Tables 1 and 2 show, we can be confident that this assumption is valid for 67.06% of GCC C compiler failures and 88.89% of Indent failures. Further study of the fault–failure relationship is out of the scope of this paper. Instead of making simplifying assumptions, for estimation of components' reliabilities we decided to consider only the failures that were caused by faults in a single component. 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.

4. Uncertainty analysis

4.1. Uncertainty analysis based on Monte Carlo simulation

For sampling from the posterior Beta and Dirichlet distributions we used the Bayesian Markov chain Monte Carlo (MCMC) method which allows drawing samples from the joint posterior density, including much more complex situations in which the likelihood and prior distribution are not conjugate. Specifically, we used Gibbs sampling [22], a special case of the Metropolis-Hastings algorithm. For the numerical results presented in this paper, we did the sampling using the WinBUGS software [29].

We checked the convergence of MCMC simulations with considerable care. The first method consisted of running multiple Markov chains simultaneously and setting sample monitors to view trace plots of the samples. If the trace plots appear to be overlapping one another, we can be reasonably confident that convergence has been achieved. We also assessed the accuracy of the posterior estimates by calculating the Monte Carlo error for each parameter. This is an estimate of the difference between the mean of the sampled values (which are used as estimates of the posterior mean for each parameter) and the true posterior mean. We ran the simulation until the Monte Carlo error for each parameter of interest was less than about 5% of the sample standard deviation. Once the convergence has been achieved accordingly to these two criteria, we ran the simulation for a further 20,000 iterations to obtain samples of the posterior distribution.

The frequency chart presented in Figure 1 gives the frequency of occurrence for different values of system reliability of GCC. The range of the reliability is $[0.999799, 0.999959]$ and the distribution is skewed to the left. We have also estimated the percentiles, i.e., certainty bands. Furthermore, we calculated the sensitivity by computing rank correlation coefficients between every parameter and system reliability. High correlation coefficient means that the parameter has a significant impact on software reliability (both through its uncertainty and its model sensitivity). Positive coefficients indicate that an increase in the parameter is associated with an increase in the reliability. As it can be seen from the sensitivity chart shown in Figure 1, 98.4% of the reliability variation is due to only 10 out of 97 parameters. All 10 of these parameters represent reliabilities. Even more, 86.6% of the system reliability variation is due to reliabilities R_{12}, R_{13}, R_1, and R_3. Of course, using different operational profile will lead to different system reliability estimate and most likely to different set of parameters that will have high impact on system reliability.

In addition to the mean reliability, we estimated several other characteristics of the system reliability distributions shown in Table 3. It is obvious that the reliability of Indent has lower mean than GCC C compiler, as well as wider range ($[0.7706, 0.8582]$). The higher variability of the reliability estimates of Indent is confirmed by the coefficient of variability which is three orders of magnitude higher that the coefficient of variability of GCC C compiler. Furthermore, the distribution of the Indent reliability is skewed slightly to the right and it has a smaller peak (see Figure 2).

	Mean	Coefficient of variability	Skewness	Kurtosis
GCC	0.999910	0.000018	-0.6861	3.8563
Indent	0.8144	0.0192	0.0227	2.985

Table 3. Reliability distributions characteristics

As it can be seen from the sensitivity chart presented in Figure 2, similarly as in case of GCC, a few parameters contribute to the most of the variability of the system reliability. Specifically, 10 out of 43 parameters are responsible for 99.6% of the variation in the reliability estimate. Also, given an operational profile, components' reliabilities have significantly higher impact on the variability of the reliability estimate than

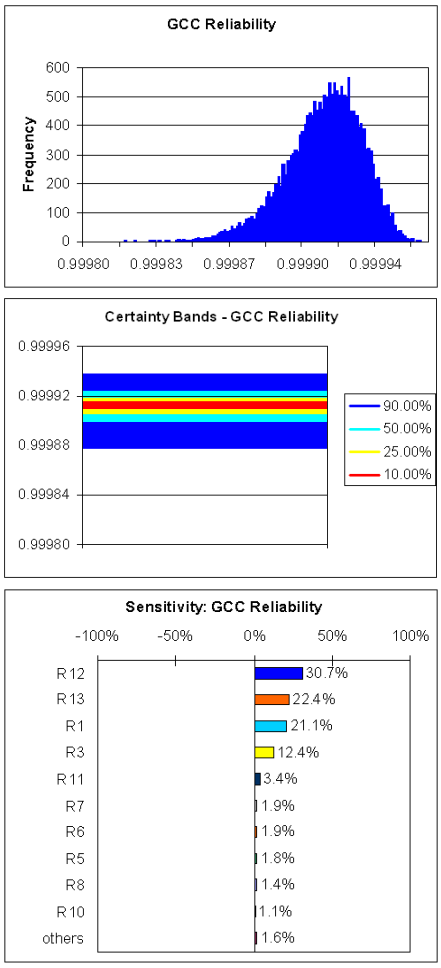

Figure 1. Uncertainty analysis for GCC

the transition probabilities. Thus, nine of the 10 parameters that contribute to 99.6% of the variance are reliability parameters. The reliabilities of only four components (i.e., R_7, R_6, R_3, and R_8) contribute 76.4% to the variance of system reliability. The only transition probability that has a significant contribution to the variance of the reliability estimate is $p_{6,end}$ with 12.8% contribution to the variance.

We also explored the accuracy of the architecture–based software reliability models by comparing the mean reliability provided by the model with the actual reliability estimated by $R = 1 - F/N$, where F is the number of system failures in N test cases[1]. The values of the actual reliability are 0.972490 for GCC C compiler and 0.8378 for Indent, which when compared to the values of the mean reliability given in Table 3 lead to errors of 2.82% and 2.80%, respectively.

4.2. Uncertainty analysis based on method of moments

Method of moments requires derivation of a symbolic close form solution for the system reliability $R = f(R_i, p_{ij})$

[1]Test cases designed to test features added to later versions, unresolved failures, and failures that led to fixing faults in multiple components are excluded both from the failed and the total number of test cases.

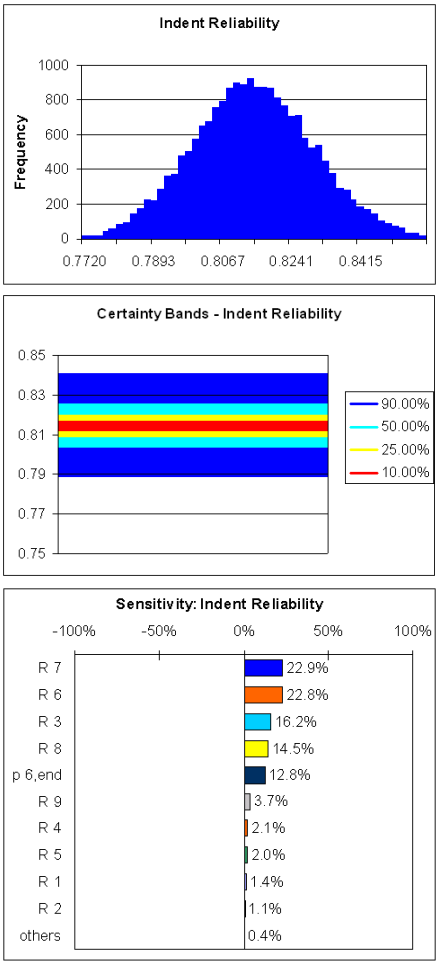

Figure 2. Uncertainty analysis for Indent

and computation of the partial derivatives. Since the complexity of the reliability expression, the number and complexity of the partial derivatives, and the number of terms in the expressions for the mean and variance increase with the number of components and non-zero transition probabilities, the method of moments does not scale well. Thus, for GCC C compiler case study, on a standard desktop (Intel IV processor and 2GB RAM), Mathematica ran out of memory while estimating the partial derivatives needed for the terms of the variance expression for the first order Taylor series approximation.

Since the Indent case study has a smaller number of components and sparser transition probability matrix, we were able to estimate the mean and variance of the system reliability for the first order Taylor series approximation. These values are as follows: $E[R] = 0.8183$, $\text{Var}[R] = 0.0012$, and the coefficient of variability is 0.0416.

5. Lessons learned and concluding remarks

In this paper we presented the uncertainty analysis of architecture–based software reliability based on empirical results obtained from two large scale software applications. Our results are based on innovative approaches to efficiently extract and more accurately analyze large amounts of empirical data. To the best of our knowledge, this is the largest and the most comprehensive empirical study ever used for assessment and uncertainty analysis of architecture–based software reliability. Based on the results presented in this paper, we have updated the list of lessons learned presented in our earlier work [11]. The first set of lessons learned is related to conducting empirical studies on architecture–based software reliability.

1. *Large quantity of data has to be extracted and analyzed.* 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.

2. *Decomposition of the system into components may not be an easy task* due to the large scale of the system and outdated documentation.

3. *Identification of faults that led to failures is not trivial.* Better format for keeping track of problem reports and source code changes, which will allow to distinguish changes made to fix faults from other changes (e.g., adding new functionality), need to be developed and adopted in practice.

The following lessons learned are related to the observations made about reliability estimations.

1. *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.

2. *Some phenomena can only be observed on large scale empirical studies.* For example, smaller case studies tend to have simpler fault–failure relationships.

3. *The architecture–based software reliability models provide accurate estimates 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.

4. *Monte Carlo simulation scales better than the method of moments* since it does not require any symbolic derivations, that is, the system reliability can be estimated numerically for each simulation run using the parameters' values sampled from the specified probability distribution functions.

5. *Monte Carlo simulation provides richer set of measures than method of moments.* These include reliability distribution function and percentiles. The results of the sensitivity ranking of the parameters accordingly to the contribution to the variance show that (1) a few parameters contribute to the most of the variation in system reliability and (2) given an operational profile, components'

reliabilities impact system reliability more significantly than transition probabilities.

As a conclusion, we believe that, similarly to more mature fields such as physics and medicine, software reliability engineering research should go through cycles of theoretical and experimental results. Thus, the experiments carried out to test a theory or explore a new domain should be followed by theoretical research that will account for the newly discovered phenomena.

Acknowledgements

This work is funded by NASA OSMA SARP under grant managed through NASA IV&V Facility in Fairmont and by NSF under CAREER grant CNS-0447715. The authors thank the contributors of GCC and Indent for their help.

References

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

[2] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles", *Proc. 23rd Int'l Conf. Software Engineering*, 2001, pp. 339–348.

[3] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing Failure: The Distribution of Program Failures in a Profile Space", *Proc. 9th ACM SIGSOFT Symp. Foundations of Software Engineering*, 2001, pp. 246–255.

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

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

[6] 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 and Dependability Symp.*, 1998, pp. 13–22.

[7] K. Goševa-Popstojanova, A. Mathur, and K. Trivedi, "Comparison of Architecture-Based Software Reliability Models", *12th Int'l Symp. Software Reliability Engineering*, 2001, pp. 22-31.

[8] K. Goševa–Popstojanova, A. Mathur, and K. Trivedi, "Comparison of Architecture-Based Software Reliability Models", *Proc. 12th Int'l Symp. Software Reliability Engineering*, 2001, pp. 22–31.

[9] K. Goševa–Popstojanova and S. Kamavaram, "Assessing Uncertainty in Reliability of Component-Based Software Systems", *Proc. 14th IEEE Int'l Symp. Software Reliability Engineering*, 2003, pp. 307–320.

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

[11] K. Goševa–Popstojanova, M. Hamill, and R. Perugupalli, "Large Empirical Case Study of Architecture–based Software Reliability", *Proc. 16th IEEE Int'l Symp. Software Reliability Engineering*, 2005, pp. 43–52.

[12] K. Kanoun and T. Sabourin, "Software Dependability of the Telephone Switching System", *Proc. 17th Int'l Symp. Fault Tolerant Computing*, 1987, pp. 236–241.

[13] S. Krishnamurthy and A. Mathur, "On the Estimation of Reliability of a Software System using Reliabilities of its Components", *Proc. 8th Int'l Symp. Software Reliability Engineering*, 1997, pp. 146–155.

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

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

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

[17] K. W. Miller et al., "Estimating the Probability of Failure when Testing Reveals no Failures", *IEEE Trans. Software Engineering*, Vol.18, No.1, 1992, pp. 33- 43.

[18] K. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution", *Proc. 1st IEEE Int'l Software Metrics Symp.*, 1993, pp. 82–90.

[19] T. J. Ostrand and E. J. Weyuker, "The Distribution of Faults in a Large Industrial Software System", *Proc. ACM Int'l Symp. Software Testing and Analysis*, 2002, pp. 55–64.

[20] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are", *Proc. ACM Int'l Symp. Software Testing and Analysis*, 2004.

[21] A. Podgurski et al., "Automated Support for Classifying Software Failure Reports", *Proc. 25th Int'l Conf. Software Engineering*, 2003, pp. 465–475.

[22] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*, Springer–Verlag, Second Edition, 2000.

[23] 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 Int'l Symp. Software Reliability Engineering*, 2001, pp. 12–21.

[24] S. Yacoub, B, Cukic, and H. Ammar, "Scenario-Based Reliability Analysis of Component-based Software", *Proc. 10th Int'l Symp. Software Reliability Engineering*, 1999, pp. 22–31.

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

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

[27] http://www.gnu.org/software/indent/indent.html

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

[29] http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml