An Analytical Approach to Architecture-Based Software Reliability Prediction

Swapna S. Gokhale[1], W. Eric Wong[2], Kishor S. Trivedi[1]*, and J. R. Horgan[2]

[1] Dept. of Electrical and Computer Engg.
Center for Adv. Comp. and Commn.
Duke University, Durham
{ssg,kst}@ee.duke.edu

[2] Bell Communications Research
445, South Street
Morristown, NJ 07960
{ewong,jrh}@bellcore.com

Abstract

Prevalent approaches to software reliability modeling are black-box based, i.e., the the software system is treated as a monolithic entity and only its interactions with the outside world are modeled. However, with the advancement and widespread use of object oriented systems design and web-based development, the use of component-based software development is on the rise. Software systems are being developed in a heterogeneous fashion using components developed in-house, contractually, or picked off-the-shelf, and hence it may be inappropriate to model the overall failure process of such systems using the existing software reliability growth models. Predicting the reliability of a heterogeneous software system based on its architecture, and the failure behavior of its components is thus absolutely essential. In this paper we present an analytical approach to architecture-based software reliability prediction. The novelty of this approach lies in the idea of parameterizing the analytic model of the software using measurements obtained from testing. To facilitate this we use a coverage analysis tool called ATAC (Automatic Test Analyzer in C), which is a part of a Software Understanding and Diagnosis System (χSuds) developed at Bellcore. We demonstrate the methodology by predicting the reliability of an application called as SHARPE (Symbolic Hierarchical Automated Reliability Predictor), which has been used to solve stochastic models of reliability, performance and performability.

*This work was supported in part by Bellcore as a core project in the Center for Advanced Computing and Communication, and by the National Science Foundation grant number EEC-9714965

1 Introduction

The impact of software structure on its reliability and correctness has been highlighted almost two decades ago [11, 13]. Prevalent approaches to software reliability modeling, however, are black-box based, i.e., the software system is considered as a whole and only its interactions with the outside world are modeled, without looking into its internal structure. Several critiques of these time-domain approaches have appeared in the literature [4, 8] and some of these include the fact that they are applicable very late in the life-cycle of the software, ignore information about testing and reliabilities of the components of which the software is made, and do not take into consideration the architecture of the software. With the advancement and widespread use of object oriented systems design and web-based development, the use of component-based software development is on the rise. The software components can be commercially available off the shelf (COTS), developed in-house, or developed contractually. Thus, the whole application is developed in a heterogeneous (multiple teams in different environments) fashion, and hence it may be inappropriate to model the overall failure process of such applications using the existing software reliability growth models (black-box approach) [1]. Thus predicting the reliability of an application early in the life-cycle, taking into account the information about its architecture, testing and reliabilities of its components, is absolutely essential.

In this paper, we present an analytical approach to architecture-based reliability prediction. The novelty of the

hybrid approach presented here lies in the fact that results from extensive testing are used to parameterize the analytical model of the application. The analytic model that appears to be appropriate is a discrete-time Markov chain (DTMC). Trace information collected during extensive testing is used to obtain branching probabilities of the DTMC. Testing of individual modules also yield coverage measurements and these are used to predict their mean value functions via the enhanced non-homogeneous Poisson process (ENHPP) model [3]. To facilitate this we use a coverage analysis tool called ATAC (Automatic Test Analyzer in C), which is a part of a Software Understanding and Diagnosis System (χSuds) developed at Bellcore. We demonstrate the methodology by predicting the reliability of an application called SHARPE (Symbolic Hierarchical Automated Reliability Predictor), which has been used to solve stochastic models of reliability, performance and performability [12]. The layout of the paper is as follows: Section 2 outlines the description and analyses of the architecture-based models, and provides a brief overview of the architecture of the application, failure behavior of the components and the methodology to predict reliability and performance, Section 3 describes the experimental set-up, Section 4 presents results and the subsequent analyses, and Section 5 concludes the paper.

2 Description and Analyses of Architecture-Based Models

The description of models to predict the reliability of an application based on its architecture requires knowledge about:

- **Architecture of the software:** This is the manner in which the different components[1] of the software interact, and is given by the intermodular transition probabilities. The architecture may also include information about the execution time (mean, variance, distribution) of each component. The architecture of sequential applications can be modeled either as a DTMC (Discrete Time Markov Chain), CTMC (Continuous Time Markov Chain) or a SMP (Semi-Markov Process), whereas concurrent applications can be modeled using SPN (Stochastic Petri Net) or DAG (Directed Acyclic Graph) [15]. The state of the application at any time is given by the component executing at that time, and the state transitions represent the transfer of control among the components. DTMC, CTMC and SMP can be further classified into irreducible and absorbing categories, where the former represents an infinitely running application, and the latter a terminating one.

- **Failure behavior of the components:** The failure behavior of the components and of the interfaces between the components, is specified in terms of the probability of failure (or reliability), constant failure rate or time-dependent failure intensity.

The architectural model can be solved and the failure behavior can be superimposed on to the solution of the architectural model to obtain reliability predictions. Analysis of the architectural model also enables us to compute various performance measures such as time to completion of the application, as well as indentify performance bottlenecks. Architectural models can be also used to analyze the impact of porting the application from one platform to the other, as well as porting from a single thread system to a distributed one.

For a particular application, depending upon the availability of information regarding the architecture and the failure behavior, the following possibilities exist, as outlined in Figure 1.

1. Information regarding the architecture as well as the failure behavior of the components is available. The architectural model along with the failure behavior of the components can then be analyzed to predict reliability and performance.

2. Only the information regarding the architecture is available, but no information regarding the failure behavior is available. Failure behavior of the components in this case can be obtained by conducting coverage testing on every component constituting the application, or by coverage measurements made during the execution of the application and a static complexity

[1] Components, modules and subsystems are used interchangeably in this paper.

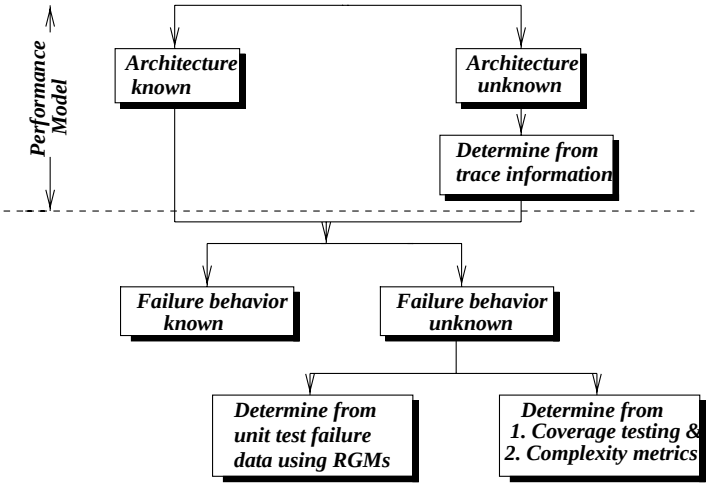

Figure 1. Possibilities regarding architecture and failure behavior

metrics based approach to predict the number of faults in the module, such as fault density [10], regression tree modeling [2], linear and non-linear regression [9], etc. An alternative way to obtain the failure behavior of a component is from the failure data collected during the unit testing of the component. An appropriate software reliability growth model can be used to fit the failure data, and to predict the residual number of faults in the component at the end of unit testing. Analysis can then be conducted to obtain reliability and performance predictions.

3. Only the information regarding the failure behavior is available, but nothing is known about the architecture of the application. The architecture of the application in this case can be determined from the trace data which can be collected by conducting extensive testing of the application. The architecture of the application along with the failure behavior can then be analyzed to predict reliability and performance.

4. Neither the information about the architecture nor the failure behavior of the components is available. The failure behavior can be determined by conducting coverage testing together with static complexity analysis as in category (2). The trace data produced during coverage testing can be used to determine the architecture of the application. Alternatively, the failure behavior of each component can also be determined from the failure data collected during unit testing of the component. Trace information can be obtained by testing the application extensively as in category (3), which can be used to determine the architecture of the application.

The application used in this study is assumed to fall in category (4), i.e., neither its architectural model nor the failure behavior is known. It is a terminating application, and we model it using a absorbing DTMC. Coverage testing of the application along with fault density analysis, gives us the failure behavior of the components via the enhanced non-homogeneous Poisson process (ENHPP) model [3]. The absorbing DTMC representing the architecture of the application is parameterized using the trace information obtained during coverage testing. We assume that the interfaces between the components are perfect and do not fail. In the subsequent subsections, we present a brief overview of the architectural model of the application, failure behavior of the components and reliability computation.

2.1 Architecture of the Application

The architecture of the application is modeled using a DTMC and we present a brief overview of DTMCs in this section. A DTMC is characterized by its one-step transition probability matrix, $\mathbf{P} = [p_{ij}]$. In the case of a DTMC with one or more absorbing states, the expected number of times the application visits state j, denoted by V_j, can be computed by solving the following system of linear equations:

$$V_j = \sum V_i p_{ij} + \pi_j(0) \quad (1)$$

where $\pi(0)$ denotes the initial state probability vector.

If t_j is the expected time spent by the application in component j per visit, then $V_j t_j$ is the expected total time spent in the component j per execution of the application. The expected completion time $\bar{t}$ of the application is given by [14]:

$$\bar{t} = \sum_{j=1}^{n} V_j t_j \quad (2)$$

2.2 Failure Behavior of the Components

The failure behavior of each component is described by a time-dependent failure intensity. Time-dependent failure intensities are determined using coverage measurements

obtained during the testing of the application and fault density approach [10] via the enhanced non-homogeneous Poisson process (ENHPP) model [3]. The ENHPP model incorporates time-varying test coverage function in its analytical formulation, and is a key step towards unifying finite failure NHPP models [1]. A brief overview of the ENHPP model is presented in this section, see [3] for details.

The enhanced non-homogeneous Poisson process (ENHPP) model states that *the rate at which the faults are detected is proportional to the product of the rate at which potential fault sites are covered and the expected number of remaining faults*. We assume that the faults are uniformly distributed, and repair is effected instantly and without the introduction of new faults. (The original formulation of the ENHPP model in [3] includes the fault detection probability, however, we assume it to be 1 in this study, and present a simplified form of the model).

Analytically, the ENHPP model is based on the following expression for its failure intensity function:

$$\lambda(t) = \frac{dm(t)}{dt} = a\frac{dc(t)}{dt} = ac^{'}(t) \tag{3}$$

or

$$m(t) = a\int_0^t c'(\tau)d\tau = ac(t) \tag{4}$$

where $m(t)$ is the expected number of faults detected by time t, and a is defined as the total number of faults which are expected to be detected given infinite testing time and complete test coverage (that is when $\lim_{t\to\infty} c(t) = 1$).

The failure intensity function $\lambda(t)$ can also be written as:

$$\lambda(t) = [a - m(t)]\frac{c^{'}(t)}{1 - c(t)} \tag{5}$$

2.3 Computation of Reliability

The reliability of module j, denoted by R_j, given the failure intensity of the module $\lambda_j(t)$, and the time spent in the module per visit t_j is [14]:

$$R_j = e^{-\int_0^{\tau_j} \lambda_j(t)dt} \tag{6}$$

where $\tau_j = V_j t_j$ is the expected total time spent in module j in a single execution of the application. V_j is the expected number of times the application visits component j during a typical execution, and can be computed using Equation (1).

Thus the reliability of the overall application, consisting of n components is given by:

$$R = \prod_{j=1}^{n} e^{-\int_0^{\tau_j} \lambda(t)dt} = \prod_{j=1}^{n} e^{-a_j c_j(\tau_j)} \tag{7}$$

3 Experimental Methodology

The experimental methodology to determine the architecture of the application and the failure behavior of its components is summarized in this section.

3.1 Selecting the Application

The Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [12] that solves stochastic models of reliability, performance, and performability was selected. This application was first developed in 1986 for three groups of users: practicing engineers, researchers in performance and reliability modeling, and students in science and engineering courses. Since then several revisions have been made to fix bugs and adopt new requirements. The current release of SHARPE is almost bug-free. It contains 35,412 lines of C code in 30 files and has a total of 373 functions. The number of lines of code in each file of SHARPE is summarized in Table 1. Each of these files is regarded as a single component for SHARPE.

3.2 Testing the Application

A suite of 735 test cases created by developers and testers for testing modifications to the existing functionality as well as new enhancements in previous releases of SHARPE was identified. A dataflow coverage testing tool, ATAC [6], was used to assess how well SHARPE could be tested by this suite. The use of ATAC focuses on three main activities: instrumenting the software to be tested, executing software tests, and determining how well the software has been tested. Instrumentation of the software occurs at compile-time, and ATAC allows large systems to be instrumented a piece at a time. Once instrumentation is complete and an executable has been built, a tester executes tests and uses ATAC to generate reports or display uncovered source code. The reports reveal the current coverage measures with respect to selected criteria[2], indicating how adequate the existing test set is and providing a view of progress during testing. In our experiments, we used block coverage, since

[2] ATAC can report coverage with respect to the function entry, block, decision, c-uses and p-uses criteria [7].

Table 1. Lines of code for components of SHARPE

Comp.	LOC	Comp.	LOC	Comp.	LOC	Comp.	LOC	Comp.	LOC	Comp.	LOC
analyze.c	946	inshare.c	1592	maketree.c	554	multpath.c	387	results.c	1322	in_qn_pn.c	1246
share.c	1977	cg.c	910	inchain.c	1203	pfqn.c	1155	bind.c	2358	bitlib.c	383
reachgraph.c	1791	sor.c	820	newcg.c	704	mpfqn.c	1142	phase.c	1957	newphase.c	1271
util.c	1119	newlinear.c	1376	cexpo.c	1267	inspade.c	880	expo.c	621	read1.c	1292
indist.c	680	symbol.c	1490	ftree.c	3560	debug.c	259	uniform.c	819	mtta.c	315

this is the most basic form of coverage that can be measured using ATAC.

A basic block, or simply a block, is a sequence of instructions that, except for the last instruction, is free of branches and function calls. The instructions in any basic block are either executed all together, or not at all[3]. A block may contain more than one statement if no branching occurs between statements; a statement may contain multiple blocks if branching occurs inside the statement; an expression may contain multiple blocks if branching is implied within the expression. Given a program and a test set, the block coverage is the percentage of the total number of blocks in the program exercised by the test set. In our case, the overall block coverage for all 735 test cases on SHARPE is 93.5% with individual coverage for each component listed in Table 2.

3.3 Determining Failure Behavior of the Components

As explained in Section 2.2 to determine the failure behavior of the components, we need to obtain the expected coverage as a function of time for each component, and the expected number of faults that would be detected from the component given infinite testing time (see Equation (3)). The expected number of faults detected given infinite testing time is determined using fault density approach [10]. To determine coverage as a function of time, we need to obtain the execution time for each test case. The execution time for each test is the difference between its starting and ending time which can be obtained from the ATAC trace file. The expected coverage is computed by running the application multiple times with the same suite of test cases but in different ordering and averaging coverages over all these runs. Stated differently, each single run gives one realization of block coverage over time; an average over a number of such runs was computed to give the expected coverage. To avoid any possible bias, the test ordering for each run was generated randomly.

[3] This definition assumes that the underlying hardware does not fail during the execution of a block.

Table 3. Block coverage with respect to time for a sample program and test set

Time (Sec)	Block coverage		
	Execution ordering t_1 followed by t_2	Execution ordering t_2 followed by t_1	Mean
0	0	0	0
1	0	0	0
2	50	0	25
3	50	25	37.5
4	50	25	37.5
5	60	60	60

We now use an example to explain the steps outlined above. Given an application with two tests, t_1 and t_2, suppose there are 20 blocks in the application: 10 of them are covered by t_1, 5 by t_2, and 3 by both. Assume also the execution time for t_1 and t_2 is 2 sec and 3 sec, respectively. Depending on which test is executed first, the block coverage may vary with time as shown in the first two columns of Table 3. Table 3 also shows the expected block coverage, which is the average of the block coverage over the runs in the first two columns.

Since testing the application repeatedly with a big test set is very costly, a trade-off is to use its block minimized subset which is the minimal subset in terms of the number of test cases that preserves the block coverage of the original test set. In our case, the number of tests is reduced from 735 to 275 because of a minimization with respect to the block coverage.

3.4 Determining the Architecture of the Application

The architecture of the application in terms of the inter-component transition probabilities is determined based on

Table 2. Block coverage for components of SHARPE

Comp.	Cov.	Comp.	Cov.	Comp.	Cov.	Comp.	Cov.	Comp.	Cov.	Comp.	Cov.
analyze.c	96.7	indist.c	98.4	pfqn.c	97.8	bind.c	98.0	inshare.c	97.5	phase.c	90.6
bitlib.c	96.0	inspade.c	99.3	reachgraph.c	75.8	cexpo.c	87.9	maketree.c	99.4	read1.c	97.9
cg.c	76.2	mpfqn.c	99.5	results.c	99.2	debug.c	69.1	mtta.c	95.5	share.c	93.2
expo.c	98.9	multpath.c	96.9	sor.c	92.6	ftree.c	93.4	newcg.c	85.2	symbol.c	97.5
in_qn_pn.c	87.8	newlinear.c	93.0	uniform.c	91.6	inchain.c	99.5	newphase.c	95.4	util.c	85.7

the following protocol: we assume that when a function A calls another function B, control is eventually transferred back to function A, except for when the application terminates successfully, or there is an error in the input specification or of some other form which causes the application to terminate abnormally. Under these situations, the control is transferred to the terminating state. Thus there are two terminating states $--$ one represents normal termination and the other represents abnormal termination. For every block, the following possibilities exist:

- If the block has neither a function call nor a return statement, then upon completion of the block, control is transferred to the next block which is physically contiguous. In this case we say that the control is transferred to another block in the same file, or at the file level of granularity we say that the control is transferred back to the same file. If the block happens to be the last block in the file, then the control is transferred back to the file which called the current file.

- If the block has a function call to one of the user-defined functions[4], which may or may not be in the same file, then the control is transferred to the first block of the called function, and we say that the control is transferred to the file in which the called function resides.

With this assumption, the branching probabilities for the control flow graph of a given application were computed based on the execution counts extracted from the ATAC trace files. The execution counts are first obtained at the finest level of granularity, i.e., the block level, and are combined later to obtain the execution counts and probabilities at the file level.

[4]System functions are treated as a single basic block.

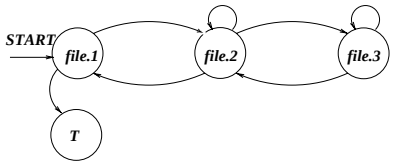

Figure 2. An example of file level transitions

We illustrate this method with the help of an example. Suppose an application consists of three files, $file.1$, $file.2$ and $file.3$: $file.1$ has two blocks, B_{11} and B_{12}, $file.2$ has three blocks, B_{21}, B_{22} and B_{23}, and $file.3$ has two blocks, B_{31} and B_{32}. The application begins by executing block B_{11}, which calls a user-defined function in $file.2$, the first block of which is B_{21}. This transfers control to $file.2$. Assume B_{21} contains no function call. Hence, the next block B_{22} gets executed upon completion of B_{21}. Suppose B_{22} calls a user-defined function in $file.3$ which begins at B_{31}. Assume B_{31} executes and passes control sequentially to B_{32}, which returns control to $file.2$ at block B_{23}, which then returns control to $file.1$ at block B_{12}. This application terminates upon completion of block B_{12}. Thus, the calling sequence at the block level is given by the following: $B_{11}, B_{21}, B_{22}, B_{31}, B_{32}, B_{23}, B_{12}$, whereas the calling sequence at the file level is $file.1, file.2, file.2, file.3, file.3, file.2, file.1, T$. The file level transitions in this scenario are as shown in Figure 2, where T indicates the terminating state of the application.

After obtaining the file level transitions, we proceed to compute the branching probabilities using execution counts. We explain the methodology used to obtain the branching probabilities by computing them for $file.1$. Suppose each block in the application is executed 10 times. Since block B_{11} is executed 10 times, and it has a call to a user-defined function in $file.2$ the first block of which is B_{21}, we say that $file.1$ calls $file.2$ 10 times. Block B_{12} is also executed 10 times, and upon completion of B_{12} the application

terminates. Thus $file.1$ transfers control 10 times to $file.2$, and 10 times to the terminating state T. Thus $file.1$ calls $file.2$ with probability 0.5, and calls the terminating state with probability 0.5. Branching probabilities can be computed for $file.2$ and $file.3$ using similar arguments.

4 Results and Analyses

The application[5] was tested using the minimal test set as explained in Section 3.3 with 40 random orderings of the test cases for each file and the coverage was measured for each run. The expected coverage as a function of time was then computed by averaging over these 40 runs for each file. The coverage for 5 individual runs and the average over 40 runs is shown in Figure 3 for $share.c$

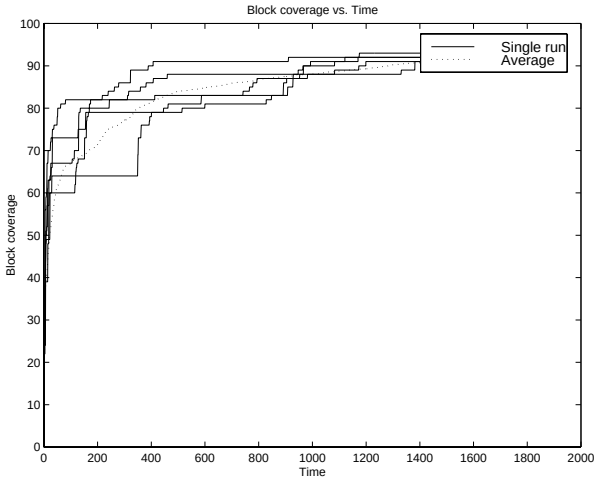

Figure 3. Block coverage for $share.c$

Given measured coverage, to obtain the reliability of an individual component, we need an estimate of the expected number of faults a_i that would be detected in component i given infinite testing time. We obtain this estimate using the fault density approach [5]. Since the application is mature, and has been in use at several hundred locations, without any known problems, we assume the fault density (FD) to be 4 per 1000 lines of code [5]. The expected number of faults in component i, denoted by a_i is given by:

$$a_i = \frac{FD * l_i}{1000} \tag{8}$$

[5] In our experiment, the application was SHARPE

where l_i is the number of lines in component i. The number of lines of code for all the components constituting the application is summarized in Table 1.

The one step transition probability matrix $\mathbf{P}$ is obtained using execution counts, which subsequently gives the visit count vector $\mathbf{V}$, as explained in Section 2.1. The execution counts for the file $bitlib.c$ are shown on the left hand side of Figure 4, and the corresponding branching probabilities are shown on the right side.

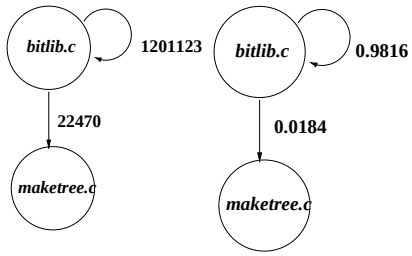

Figure 4. Execution counts and branching probabilities for $bitlib.c$

The branching probabilities are computed on a per block basis, so it is reasonable to assume that that a single block is executed per visit to a file. Thus the expected time spent by the application in a component per visit, is the expected time taken to execute a block (EB), which can be computed as follows:

$$EB = \frac{TT}{TB} \tag{9}$$

where TT is the total time taken to execute 735 test cases, and TB are the total number of block executions. The total time taken to execute 735 test cases is 5582 sec, and the total number of block executions is 970322031. Hence the expected execution time of a block, from Equation (9) is 0.0000057257. The expected time spent by the application in each component is summarized in Table 4, and the mean value function of each component is summarized in Table 5. The component having the highest mean value function is the reliability bottleneck, whereas the component with the maximum expected time is the performance bottleneck. Thus, from Table 4, and Table 5, $pfqn.c$ is the performance as well as the reliability bottleneck.

Table 4. Expected time per component of SHARPE

Comp.	Exp. time	Comp.	Exp. time	Comp.	Exp. time	Comp.	Exp. time	Comp.	Exp. time	Comp.	Exp. time
analyze.c	0.0091	inshare.c	0.0033	maketree.c	0.0013	multpath.c	0.0002	results.c	0.0053	in_qn_pn.c	0.0008
share.c	0.0210	cg.c	0.0035	inchain.c	0.0152	pfqn.c	3.4439	bind.c	0.1037	bitlib.c	0.0010
reachgraph.c	0.2011	sor.c	0.0806	newcg.c	0.0003	mpfqn.c	0.0010	phase.c	0.0450	newphase.c	0.0005
util.c	1.0726	newlinear.c	0.0005	cexpo.c	0.2908	inspade.c	0.0014	expo.c	0.0125	read1.c	0.6076
indist.c	0.0015	symbol.c	0.0193	ftree.c	0.7439	debug.c	0.0001	uniform.c	0.1149	mtta.c	0.0047

Table 5. Mean value function (MVF) per component of SHARPE

Comp.	MVF	Comp.	MVF	Comp.	MVF	Comp.	MVF	Comp.	MVF	Comp.	MVF
analyze.c	2.31e-06	inshare.c	1.96e-07	maketree.c	0.00	multpath.c	0.00	results.c	2.19e-06	in_qn_pn.c	1.91e-07
share.c	1.37e-05	cg.c	0.00	inchain.c	1.18e-05	pfqn.c	0.0011	bind.c	2.69e-05	bitlib.c	0.00
reachgraph.c	0.00	sor.c	7.69e-06	newcg.c	0.00	mpfqn.c	3.64e-07	phase.c	0.00	newphase.c	0.00
util.c	2.78e-04	newlinear.c	8.37e-08	cexpo.c	2.76e-06	inspade.c	6.48e-07	expo.c	7.38e-07	read1.c	3.36e-04
indist.c	1.81e-07	symbol.c	0.00	ftree.c	6.16e-04	debug.c	0.00	uniform.c	0.0	mtta.c	1.07e-06

The expected time to completion of the application, as computed from Equation (2) is 6.80 sec. The expected completion time can also be computed empirically, since it takes 5582 sec to execute 735 test cases, and is 7.54 sec. The reliability of the application from Equation (7) is 0.9903. Whereas performance analysis was not the primary objective of this experiment (the primary objective was reliability prediction), comparison of the mean time to completion of the application calculated using our approach, with the empirical mean time to completion served as a sanity check, to ensure that the results obtained using this approach are within reasonable limits. The discrepancy between the empirical expected time to completion and the one computed using our approach could be due to the fact that the empirical value of the time to completion will be influenced by the load on the system, and a large variation is possible in this value.

To determine the effect of the variation in fault density of the entire application on the reliability estimate obtained using this approach, we computed the reliability for various values of the fault density. Figure 5 shows reliability of the application vs. fault density. As can be seen from this figure, the reliability of the application drops with increasing fault density.

Typically, in a component based software development scenario, some of the components are developed in-house, while some are picked off the shelf. The failure behavior of the components picked off the shelf is certified, and we

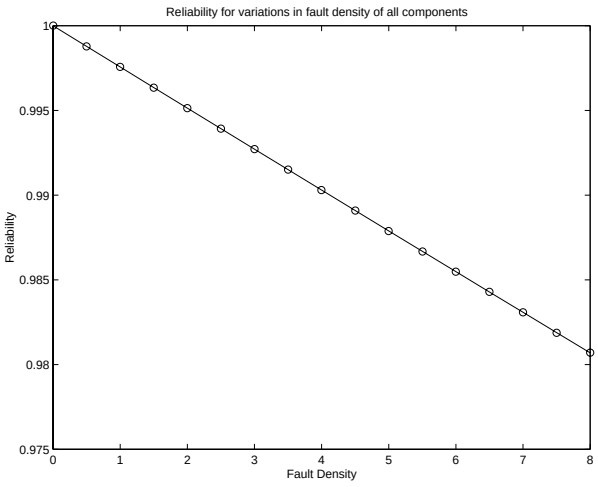

Figure 5. Effect of variations in fault density of the entire application

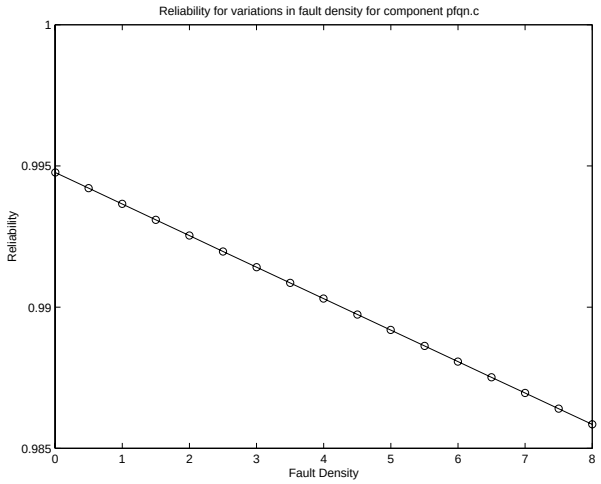

Figure 6. Effect of variations in fault density of $pfqn.c$

have information regarding the failure behavior of the components that are developed in-house. In addition, the architecture of the entire application (in terms of interactions among the various components) is known. We would like to assess the sensitivity of the reliability estimate to the variations in the failure behavior of the off-the-shelf component. The failure behavior of the component can vary due to the variations in its fault density. We assume that the file $pfqn.c$ was picked off the shelf. The reliability of the application for variations in the fault density of $pfqn.c$, is shown in Figure 6. As expected, the reliability decreases with increasing values of fault density.

5 Conclusions

In this paper we have described an analytical approach to architecture-based software reliability prediction. The parameters of the analytical model (architecture of the application in terms of the branching probabilities, and failure model of its components) are determined experimentally by testing the application using the regression test suite, and obtaining trace information using a coverage measurement tool called ATAC. The information about the architecture and the failure behavior is then analyzed to obtain reliability predictions. Sensitivity of the reliability estimate thus obtained, to variations in the values of the parameters was also determined. The reliability estimates obtained using this approach will depend heavily on the test suite used. Hence the test suite used in this approach, should as far as possible reflect the actual usage of the application. Further experiments to validate the reliability predictions, and test the influence of the assumptions on the estimates are currently underway.

6 Acknowledgments

The authors wish to thank Dr. Robin Sahner for providing the regression test suite for SHARPE. The authors also wish to acknowledge the χSuds team at Bellcore, especially Saul London, for his invaluable help and guidance in the use of ATAC.

References

[1] W. Farr. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Reliability Modeling Survey, pages 71–117. McGraw-Hill, New York, NY, 1996.

[2] S. Gokhale and M. R. Lyu. "Regression Tree Modeling for the Prediction of Software Quality". In *Proc. of ISSAT '97*, pages 31–36, Anaheim, CA, March 1997.

[3] S. Gokhale, T. Philip, P. N. Marinos, and K. S. Trivedi. "Unification of Finite Failure Non-Homogeneous Poisson Process Model through Test Coverage". In *Proc. Intl. Sypmosium on Software Reliability Engineering (ISSRE '96)*, pages 289–299, White Plains, NY, October 1996.

[4] D. Hamlet. "Are We Testing for True Reliability ?". *IEEE Software*, 13(4):21–27, July 1992.

[5] M. Hecht, D. Tang, H. Hecht, and R. W. Brill. "Quantitative Reliability and Availability Assessment for Critical Systems Including Software". In *Proc. of Computer Assurance '97*, pages 147–158, Gatheirsburg, Maryland, June 1997.

[6] J. R. Horgan and S. London. "ATAC: A Data Flow Coverage Testing Tool for C". In *Proc. of Second Symposium on Assessment of Quality Software Development Tools*, pages 2–10, New Orleans, Louisiana, May 1992.

[7] J. R. Horgan and S. A. London. "Dataflow Coverage and the C Language". In *Proc. of the Fourth Annual Symposium on Testing, Analysis and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991.

[8] J. R. Horgan and A. P. Mathur. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Testing and Reliability, pages 531–566. McGraw-Hill, New York, NY, 1996.

[9] T. M. Khoshgoftaar, B. B. Bhattacharyya, and G. D. Richardson. "Predicting Software Errors, During Development, Using Nonlinear Regression Models: A Comparative Study". *IEEE Trans. on Reliability*, 41(3):390–395, September 1992.

[10] M. Lipow. "Number of Faults per Line of Code". *IEEE Trans. on Software Engineering*, SE-8(4):437–439, July 1982.

[11] D. L. Parnas. "The Influence of Software Structure on Reliability". In *Proc. 1975 Int'l Conf. Reliable Software*, pages 358–362, Los Angeles, CA, April 1975.

[12] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, Boston, 1996.

[13] M. L. Shooman. "Software Reliability: A Historical Perspective". *IEEE Trans. on Reliability*, R-33(1):48–54, April 1984.

[14] K. S. Trivedi. *"Probability and Statistics with Reliability, Queuing and Computer Science Applications"*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[15] A. T. Wei and R. H. Campbell. "Construction of a Fault Tolerant Real-Time Software System". Technical Report UIUCDCS-R-80-1042, U. of Illinois, Urbana-Champaign, December 1980.