An Analytical Approach to Architecture-Based Software Performance and Reliability Prediction

Swapna S. Gokhale[1], W. Eric Wong[2], J. R. Horgan[3] and Kishor S. Trivedi[4]
[1] Dept. of CSE, Univ. of Connecticut, Storrs, CT 06269
[2] Dept. of CS, Univ. of Texas at Dallas, Richardson, TX 7508
[3] Applied Research, Telcordia Technologies, Morristown, NJ 07960
[4] Dept. of ECE, Duke Univ., Durham, NC 27708

Abstract

Conventional approaches to analyze the behavior of software applications are black box based, that is, the software application is treated as a whole and only its interactions with the outside world are modeled. The black box approaches ignore information about the internal structure of the application and the behavior of the individual parts. Hence they are inadequate to model the behavior of a realistic software application which is likely to be made up of several interacting parts. Architecture–based analysis, which seeks to assess the behavior of a software application taking into consideration the behavior of its parts and the interactions among the parts is thus essential. Most of the research in the area of architecture–based analysis has been devoted to developing analytical models, with very little if any effort being devoted to how these models might be applied to real software applications. In order to apply these models to software applications, methodologies must be developed to extract the parameters of the analytical models from information collected during the execution of the application. In this paper we present an experimental methodology to extract the parameters of architecture–based models from code coverage measurements obtained during the execution of the application. To facilitate this we use a coverage analysis tool called ATAC (Automatic Test Analyzer in C), which is a part of Telcordia Software Visualization and Analysis Toolsuite (TSVAT) developed at Telcordia Technologies. We demonstrate the methodology by predicting the performance and reliability of an application called SHARPE (Symbolic Hierarchical Automated Reliability Predictor), which has been widely used to solve stochastic models of reliability, performance and performability.

Keywords

Software Performance, Software Reliability, Software Architecture Markov Chain, Semi–Markov Process

1 Introduction

The size and complexity of computer systems has increased more rapidly in the past decade, than our ability to design, test, implement and maintain them. Computer systems are being increasingly used in various active (controlling), and passive (monitoring) applications, and the trend will surely continue in the future. Computer system failures make newspaper headlines because at best they inconvenience people (e.g., malfunctions of home appliances), cause economic damage (e.g., interruptions of banking services), and in the extreme cases cause deaths (e.g., failures of flight control systems or medical software). The computer industry has seen uneven progress. With the steadily growing power and reliability of the hardware, software reliability has been identified as a major stumbling block in the realization of highly dependable computer systems. When lives and fortunes depend on software, assurance of its quality becomes an issue of critical concern.

Conventional approaches to analyzing the performance and reliability of software applications treat the software application as a whole and only its interactions with the outside world are modeled [11]. One of the most notable drawbacks of these approaches is that they ignore the internal structure of the application and the performance and reliability behavior of the various parts of which the application is made up of [18, 24]. Even a moderate sized software application is likely to be developed using a "divide and conquer" strategy and made up of several interacting parts. As a result, conventional approaches which treat the application as a whole are inadequate to model the behavior of even moderate sized applications. A recognition of the inadequacy of conventional approaches is evident from the fact that a few research efforts have addressed the issue of characterizing the behavior of modular software applications since the 1970s [7, 31, 32, 33, 36, 38, 39, 40]. The advent of component technologies and object–oriented programming holds the promise of realizing the vision of assembling software applications from systematically developed reusable software components. This has generated renewed interest in "architecture–based analysis" which aims to characterize the performance and reliability behavior of software applications based on the behavior of the "components" and the "architecture" of the application [10, 15, 13, 17, 19, 25, 28, 29, 44]. We note that the notion of "architecture" and "components" is well defined only in the context of applications that are assembled from software components using one of the component technologies such as Microsoft's COM/DCOM [1] or CORBA component model [2] or Sun Microsystems' JavaBeans [4] and Enterprise JavaBeans [3]. The notion of architecture and components is not clearly defined in the context of applications which are built ground up. However, the use of architecture–based analysis which essentially characterizes the behavior of a software application in terms of the behavior of its parts and interactions among the parts is not limited to applications developed using the component–based approach. In fact, architecture–based

analysis is valuable towards analyzing how different pieces of the application interact and contribute to the overall application reliability even for an application that is built ground up.

The existing approaches to architecture–based analysis can be classified into three categories, namely, state–based [7, 27, 31], path–based [25, 47, 41] and additive [46] as proposed by Goseva–Popstojanova [16]. Of the three types of approaches, state–based approaches have received a maximum degree of attention. State–based models use the control flow graph to represent software architecture and evaluate software reliability analytically. They assume that the transfer of control among the components follows a Markov property. Software architecture is modeled using a discrete time Markov chain (DTMC) [7], continuous time Markov chain (CTMC) [27, 31], or a semi–Markov process (SMP) [26]. Most of the research efforts in the area of state–based models have focused on the development of theoretical models. Very little energy has been devoted to the question of how to apply these models to real software applications. In order to apply these models to real software applications, the parameters that drive these models need to be obtained. These parameters describe the architectural behavior of a application and the failure characteristics of its components. These parameters can be obtained from a variety of sources. Unlike black box approaches which can be applied only during the testing phase of the application (which is one of the drawbacks of the black box approaches [18, 24]), architecture–based analysis can be applied as early as the design phase of the software application. A software application that is developed in an ideal manner will go through a first round of architecture–based analysis during the design phase. When architecture–based analysis is conducted during the design phase, the architecture of the application may be obtained from expert opinion, from prior experience with a similar product, from the previous release of the same product, by simulation or may be "guestimated". It is unlikely that the failure behavior of the components is available during the design stage except for components which are being reused or are picked off the shelf. If the component is an off the shelf component its reliability may be certified [19]. In the design phase, architecture–based analysis can be used to answer questions such as: (i) which components are critical to the performance and reliability of the application, and (ii) how is the application reliability influenced by the performance and reliabilities of individual components? If the software application is going to be assembled from a collection of components, then answers to such questions can help the designers to make decisions such as which components should be picked off the shelf, and which components should be developed in house. If the software application is going to be developed ground up, then such answers can help the designers in making decisions such as which components should be developed by experienced developers. As the application progresses through the development and integration phases, additional data may be available which could be used to refine the parameters representing the architecture as well as the failure behavior. During

the testing phase, measurements obtained during the execution of the application can be used to refine the architecture–based models further. In order to use the execution information generated during the testing of the application to refine the architectural models, methodologies must be developed to record and process the execution information. The methodology capable of using execution information for architecture–based analysis will not only be useful in refining architectural models in the testing phase, but will also be useful towards applying such analysis to a software application that has already been deployed and is operational. Applying architecture–based analysis to an operational software application will enable us to abstract the current characteristics of its architecture as well as its components into a model. These current characteristics may deviate significantly from the characteristics expected in the design phase and also from the characteristics that existed just prior to deployment. These deviations may be a result of the inevitable evolution of the software application. Abstracting the current characteristics into a model can be used to answer questions such as: (i) what will be the impact on application performance and reliability if component X is to be replaced by component Y?, and (ii) in order to improve the application performance and reliability which components should be targeted? In addition, the analytical model can also be useful in analyzing the effect of porting the application between different platforms. If the application was not developed using a component–based paradigm, but needs to be adapted to the component world, then model–based analysis can enable informed decisions about which pieces should be developed in–house, and which pieces should be reused by pointing to those pieces that are critical to application performance and reliability.

The key to refining architecture–based models in the testing phase or to apply the analysis in the operational phase is in generating and processing execution information to extract the parameters of architecture–based models. The generation and processing of execution information will be influenced by several factors such as: (i) whether the application was developed ground up, or assembled using a component–based approach, and (ii) whether the source code of the application is completely or partially available. The current paper presents an experimental approach to generate and process the execution information in order to extract the parameters of architecture–based models for applications developed ground up and with complete access to the source code. This methodology needs to be adapted in order to be applied to applications assembled using software components, since such applications tend to differ from the applications developed ground up. In Section 3 we briefly describe how the methodology may be adapted to apply to component–based software applications. For applications developed ground up, the notion of a component is not very well–defined and is largely a matter of tradeoff between number of components, their size and how easily the necessary information can be extracted. We designate a single source file as an individual component of the application. This designation was motivated by the hope that a file would host a group of related

functions which are likely to be placed in a single component in the componentized version of the application. This assumption holds reasonably well for the application we have used for the demonstration of our experimental approach. However, we would like to note that our experimental methodology is not limited by this designation. In fact, our experimental methodology can extract parameters, even if an individual component is assumed to consist of a collection of a small number of source statements. We elaborate on this issue further in the paper (See Section 3.2). We parameterize the analytical model of the application using coverage measurements generated during the execution of the application. The novelty of the approach lies in its integration of two distinct yet important areas, namely, state–based software performance and reliability models and coverage analysis. The experimental approach thus presents an initial step towards addressing the important issue of how one might apply analytical architecture–based software reliability and performance analysis to a real software application.

The state–space approach used to model the architecture of the application is a discrete time Markov chain (DTMC). Coverage measurements generated during the execution of the application are used to determine the failure behavior of the components via the enhanced non homogeneous Poisson process model [14]. Coverage measurements are also used to extract the intercomponent transition probabilities of the architecture of the application. Our experimental approach is facilitated by a coverage analysis tool called ATAC (Automatic Test Analyzer in C) which is a part of the Telcordia Software Visualization and Analysis Tool Suite (TSVAT) [22]. We demonstrate the methodology by predicting the reliability and performance of an application called SHARPE (Symbolic Hierarchical Automated Reliability Predictor). SHARPE has been used to solve stochastic models of reliability, performance and performability [37].

The layout of the paper is as follows: Section 2 outlines the description and analyses methods of state–based models and provides a brief overview of the various architectural models, 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. Section 5 concludes the paper and presents directions for future research.

2 Description and analyses of state-based models

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

- **Architecture of the application:** This is the manner in which the different components[1] of the software interact, and is given by the intermodular transition probabilities. The

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

architecture may also include information about the execution time (mean, variance, distribution) of each component. The architecture of the application can be modeled either as a DTMC (Discrete Time Markov Chain) [7], CTMC (Continuous Time Markov Chain) [27], SMP (Semi-Markov Process) [26], DAG (Directed Acyclic Graph) [45] or a SPN (Stochastic Petri Net). 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, SMP and SPN can be further classified into irreducible and absorbing categories, where the former represents an infinitely running application, and the latter a terminating one. The architecture of the application provides the performance model of the application. Analysis of the performance model can be used to obtain performance predictions such as time to completion of the application, and identify performance bottlenecks. Performance models can also be used to analyze the effect of porting the application to a different platform, or porting it from a sequential platform to a distributed one.

- **Failure behavior of the components/interfaces:** Failure may occur during the execution of any component or during the control transfer between two components. The failure behavior of the components may be specified in terms of the probability of failure (or reliability), time–independent failure rate or time–dependent failure intensity. It is a widely known fact that interface failures or failures that occur during the transfer of control between two components should be considered separately from individual component failures. However, very little information is available on how the parameters representing the failure behavior of the interfaces may be estimated. The failure behavior of the interfaces may also be specified in terms of the probability of failure, time–independent failure rate or time–dependent failure intensity.

The information about the architecture of the application and the failure behavior of its components and the interfaces between the components can be combined in the following two different ways to predict application performance and reliability.

- **Composite method:** The architecture of the application can be combined with the failure behavior of the components and the interfaces into a composite model which can then be analyzed to predict the performance and reliability of an application. We will refer to this method of performance and reliability prediction as “Composite Method”.

- **Hierarchical method:** The other possibility is to solve the architectural model and superimpose the failure behavior of the components and the interfaces on to the solution of the architectural model, to predict reliability. Solution of the architectural model provides the performance metrics for the application. We refer to this method as “Hierarchical Method”.

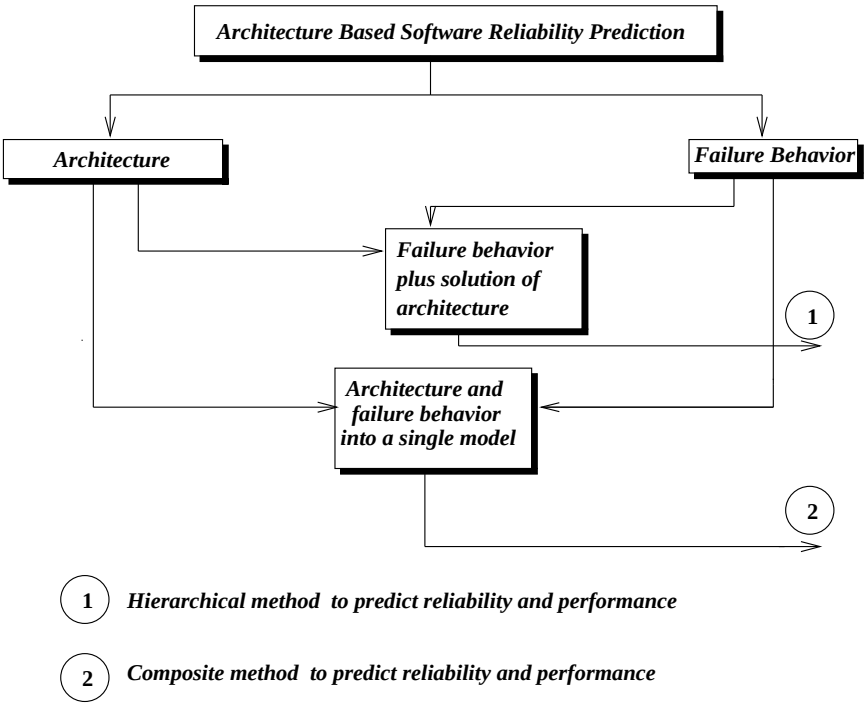

Figure 1: Description and analyses of architecture–based models

The information required for the description of state–based models, and the different ways in which this information can be analyzed in order to predict performance and reliability are depicted in Figure 1. An exhaustive discussion of the composite and hierarchical approaches to predict the reliability of an application is presented in [15].

The specific analytical model we use has the following features:

- Architecture: The architecture of the application is assumed to be modeled by a discrete time Markov chain (DTMC).
- Failure behavior: Failure behavior of the components is assumed to be characterized by time–dependent failure intensity. Time–dependent failure intensity was chosen to characterize the failure behavior of the components because of its ability to express the dependence of the failure behavior on the code characteristics and how the code is being used. In addition, time–dependent failure intensity can also capture intra–component dependence via cumulative execution time. We elaborate on these issues in Section 2.2 and Section 2.3 respectively. We assume that the interfaces do not fail. We note that this

assumption is not necessary to ensure model tractability. The model can be extended to accommodate unreliable interfaces. However, it is not clear how the parameters representing the failure behavior of the interfaces may be estimated when an interface consists of items such as global variables, parameters and files. Integration testing proposed in [8, 9] may hold a lot of promise for estimating interface reliabilities and is the topic of our future research.

- Solution method: We use the hierarchical method of solution. As explained in Section 2.3, analysis using the composite method is not feasible for this particular combination of architectural model and failure behavior.

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, $\mathbf{P} = [p_{ij}]$. The state probability vector of the DTMC at time step n is denoted by $\pi(n)$.

$$\pi = \pi.\mathbf{P} \tag{1}$$

$$\pi.\mathbf{e} = 1 \tag{2}$$

where $\mathbf{e} = (1, 1, \ldots, 1)^T$ and the superscript T denotes the transpose.

From the point of view of modeling the architecture of software applications, DTMCs can be classified into the following two categories:

- Irreducible: A DTMC is said to be irreducible if every state can be reached from every other state. An irreducible DTMC can be used to model the architecture of an infinitely running application.

- Absorbing: A DTMC is said to be absorbing, if there is at least one state i, from which there is no outgoing transition. Starting from the start state, a DTMC upon reaching an absorbing state is destined to remain there forever. An absorbing DTMC can be used to model the architecture of a terminating application or the one that operates on demand.

In the case of a DTMC with one or more absorbing states, the expected number of times the process visits state j, denoted by V_j, can be computed by solving the following system of linear equations:

$$V_j = \sum V_i p_{ij} + \pi_j(0) \tag{3}$$

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

We can use the DTMC analysis presented in this section to obtain performance measures such as time to completion of the application, identify performance bottlenecks, determine the effect of changes in the workload, and determine the effect of changes to a particular module, as follows:

- **Time to completion:** If t_j, the expected time spent by the application in component j per visit is known (t_j can either be obtained experimentally or may be known *a priori*), then $V_j t_j$ is the expected total time spent in the component j in a typical execution of the application. For an application with n components, the expected completion time $\bar{t}$ of the application is given by [42]:

$$\bar{t} = \sum_{j=1}^{n} V_j t_j \tag{4}$$

- **Performance bottlenecks:** $argmax\{V_j t_j\}$ is the performance bottleneck of the application. Thus, we note that neither V_j, which is the expected number of visits to component j during a single execution of the application, nor t_j, which is the expected time spent by the application in component j per visit, are individually sufficient to determine the performance bottleneck, but their product $V_j t_j$ is. This definition of performance bottleneck is applicable only in the context of a terminating application.

- **Changes in the workload:** A workload pattern is determined by a set of intercomponent transition probabilities. For a software application the set of intercomponent transition probabilities will be affected by the operational profile [35]. Upgrades to the software may introduce new features in an application which may lead to a change in how the existing features are used. This may render an existing estimate of the operational profile invalid. When the operational profile changes, a new set of intercomponent transition probabilities may be obtained by executing the application against the sample test cases drawn from the new operational profile. An estimate of the performance of the application could also be obtained during the process of estimating intercomponent transition probabilities. However, the process of testing and estimation will have to be repeated every time the operational profile changes. In a well–designed application where the interactions among the components is limited it may be possible to determine which interactions are impacted by a change in the operational profile by consulting the experts. Model–based analysis may then be used to determine the sensitivity of the performance

and reliability estimates to the changes in the intercomponent transition probabilities among the components which are likely to be impacted by the change in the operational profile. An analytical model will enable us to obtain the performance and reliability estimates for various values of intercomponent transition probabilities among the impacted components without actually having to estimate the values of the probabilities. Estimation of the intercomponent transition probabilities and performance metrics by extensive testing can be lengthy both in terms of computation time and resources.

- **Changes to a module:** If a single module changes while preserving its interaction with the other modules (as determined by the intercomponent transition probabilities), and all the other modules remain unchanged, the overall performance can be predicted by simply measuring the execution time per visit of a changed component. Thus, if component c changes, the time to completion of the application is given by:

$$\bar{t}^* = (\sum_{j=1, j\neq c}^{n} V_j t_j) + V_c t_c^* \tag{5}$$

In Equation (5), t_c^* is the new execution time per visit for component c. We note that the expected number of visits to the component, denoted by V_c, which depends on the intercomponent transition probabilities remains unchanged. Model–based analysis can thus provide the capability to assess the impact of a change to a module by estimating the values of the parameters only for the changed module. If the performance of the application had been determined empirically by running test cases against the application and averaging over all the runs, then assessing the impact of a change to a module would require us to repeat the empirical procedure. Repetition of the empirical procedure is likely to be more costly than estimating the values of the parameters for the changed module.

2.2 Failure behavior of the components

Intuitively, the failure behavior of a component will depend on the code characteristics of the component and how the component is being used. Empirical evidence has supported the conjecture that the failure behavior is highly correlated with code coverage [5, 6, 12, 34]. Code coverage can thus be a useful measure to capture how the component is being used. Hence we use time–dependent code coverage as a parameter in a statistical model to estimate the failure intensity of the component. The characteristics of the code can be captured by estimating the number of faults in the component based on software metrics. We estimate the number of faults in each component based on the number of lines of code in the component using the fault density approach [30].

The enhanced non homogeneous Poisson process (ENHPP) model [14] relates the failure intensity of a component to the expected number of faults residing in the component and its time–dependent coverage behavior. This relationship is exemplified by the following expression:

$$\lambda(t) = ac^{'}(t) \tag{6}$$

where a is the expected number of faults present in the component and $c^{'}(t)$ is the first derivative of the expected coverage behavior $c(t)$ of the component.

The reliability of a component j, denoted by R_j, given the failure intensity of the component, and the expected time spent in the module γ_j (where $\gamma_j = V_j t_j$), is:

$$R_j = e^{\int_0^{\gamma_j} \lambda_j(\theta) d\theta} \tag{7}$$

From Equation (6), Equation (7), can be written as:

$$R_j = e^{-\int_0^{\gamma_j} a_j {c_j}^{'}(\theta) d\theta} = e^{-a_j c_j(\gamma_j)} \tag{8}$$

2.3 Method of analyses

In this section, we motivate the choice of the hierarchical method of analyses for an application with architecture modeled by a DTMC, and the failure behavior modeled by the time–dependent failure intensity with the help of an example. When the architecture of the application is modeled by a discrete time Markov chain (DTMC) and the failure behavior of the components are characterized by their reliabilities a composite method of analyses is possible as discussed by Cheung [7]. However, characterizing the failure behavior of a component by its reliability cannot take into consideration how the component is being used. As discussed earlier, time–dependent failure intensity expressed as a composition of the number of faults in the component and time–dependent coverage behavior can take into consideration both the factors that affect component failures, namely, the code characteristics and the how the code is being used. As a result, we characterize the failure behavior of the individual components by their time–dependent failure intensities.

Consider an application with three components. The architecture of the application is described by the DTMC shown on the left in Figure 2. Let the failure intensity of component j be denoted by $\lambda_j(t)$, and γ_j be the cumulative time spent by the application in component j from the beginning of system operation. Combining the architecture and the failure behavior of the application, gives rise to a composite reliability model as shown on the right in Figure 2, where the state F indicates the failure of the application. However, since successive sojourns in a given state are not in general contiguous, the reliability model shown on the right in Figure 2

is a complex stochastic process (it is not a non–homogeneous CTMC nor is it a semi–Markov process) that cannot be analyzed using composite methods.

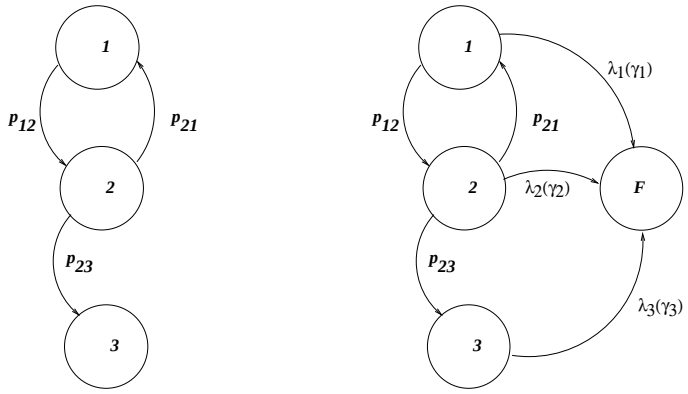

Figure 2: Composite reliability model

The analysis is much simpler using hierarchical method of analyses, by which the reliability of the application, denoted by R is given by:

$$R = \prod_{j=1}^{3} R_j \tag{9}$$

where R_j is the reliability of module j.

Thus, the reliability of the overall application is given by:

$$R = \prod_{j=1}^{3} e^{-\int_0^{\gamma_j} \lambda(\theta) d\theta} \tag{10}$$

Equation (10) can be generalized to compute the reliability of an application with n components, and is given by:

$$R = \prod_{j=1}^{n} e^{-\int_0^{\gamma_j} \lambda(\theta) d\theta} \tag{11}$$

From Equation (8), Equation (11) can be written as:

$$R = \prod_{j=1}^{n} e^{-a_j c_j (V_j t_j)} \tag{12}$$

Equation (10) assumes inter–component independence [25], that is, the execution of one component does not in any way affect the failure behavior of any other component, an assumption which underlies most state–based software reliability and performance models. The execution of one component could affect the failure behavior of another component due to error propagation via data flow. Hence error propagation via data flow could be an important factor influencing application reliability. The path–based model proposed by Singpurwalla *et. al.* [41] considers error propagation via data flow along with the control flow of the application. However, path–based models are unable to account for the infinite number of paths that might exist due to the existence of loops in the control flow graph. State–based models can analytically account for the infinite number of paths. Developing state–based models which account for error propagation via data flow is a subject of our future research. The effects of error propagation due to data flow can be quantified using fault insertion testing [21, 43].

Modeling the failure behavior of a component using time–dependent failure intensity enables us to capture intra–component dependence [25] as explained below. Intra–component dependence can arise for example, when a component is invoked more than once in a loop. Reliability values assuming intra–component independence are in general pessimistic as pointed out by Krishnamurthy *et. al.* [25]. They resolve the issue by collapsing multiple executions of the same component into k occurrences where k is defined as the degree of independence.

If the failure behavior of the individual components is modeled by a time–dependent failure intensity then it is possible to capture intra–component dependence via cumulative execution time spent in a component. The reliability value obtained assuming intra–component independence is in general pessimistic as compared to the reliability value which is based on the cumulative time spent in the component. We illustrate this with the help of an example as follows: Consider a component whose failure behavior is captured by the failure intensity function $\lambda(t) = 34.05 * 0.0057 * e^{(-0.0057*t)}$. We assume that during a particular run, the component is visited twice, and 50 time units are spent in the component per visit. Thus a total of 100 time units is spent in the component during this particular execution. The failure intensities assuming intra–component independence, and intra–component dependence captured via the cumulative execution time approach are as shown in the Figure 3. The reliability of the component assuming intra–component independence is 0.84, and intra–component dependence captured via the cumulative time approach is 0.86. Time–dependent failure intensity thus provides an ability to capture intra–component dependence.

Equation (11) indicates that the reliability of the application $R \rightarrow 1.0$ as γ_j's $\rightarrow 0$, for all j. This is consistent with the software reliability model proposed by Singpurwalla *et. al.* [41] which can be expressed as the following:

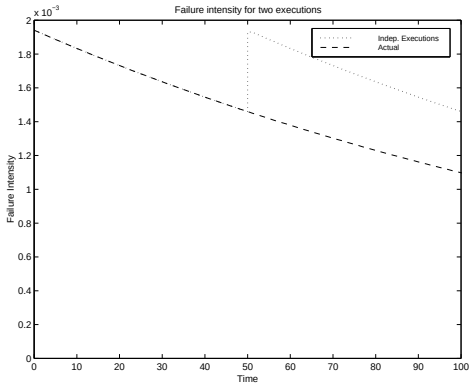

Figure 3: Failure intensity – independent execution and cumulative time

$$P(\varepsilon|H) = P(\varepsilon|\Theta)P(\Theta|H) \tag{13}$$

In Equation (13) $P(\varepsilon|H)$ is the reliability model of the software conditional to the background information H. Θ is known as the parameter space of the reliability model and is interpreted as a device for summarizing the background information H. In the context of Equation (11), the background information of the application can be viewed to have two possibilities: (i) the application is executed, (ii) the application is not executed. Thus, the parameter space Θ intended to capture the background information in this case consists of γ_j's. Thus, γ_j is equal to zero for all j if the application is not executed at all resulting in a reliability of 1. At least one γ_j will be greater than zero if the application is executed resulting in a reliability value of less than or equal to 1.0.

Equation (12) expresses the overall reliability of an application in terms of the failure behavior of its individual components and architectural characteristics. This approach is very valuable to identify reliability bottlenecks, as well as the effect of changes in a single module, as discussed below:

- **Reliability bottlenecks:** Reliability bottleneck is given by $argmax\{a_j c_j (V_j t_j)\}$. Thus, the code characteristics of the component and the coverage behavior which captures how the component is being used, together determine the reliability bottleneck. This definition of reliability bottleneck is applicable only in the context of terminating applications.

- **Changes to a module:** If a single module is changed while preserving all the other aspects of the application, namely, the intercomponent transition probabilities, and failure behavior of the other unchanged components, the reliability of the application is given by:

$$R^* = \left(\prod_{j=1, j \neq c}^{n} e^{-a_j c_j (V_j t_j)} \right) e^{-a_c^* c_c^* (V_c^* t_c^*)} \quad (14)$$

where $a_c^* c_c^* (V_c^* t_c^*)$, is the mean value function of the changed component. We note that model–based analysis renders itself well to evaluate the impact of a change to a module by estimating the parameters of the changed module. If the reliability of the application were to be determined empirically, then a change to a module would require the repetition of the empirical procedure in order to evaluate the impact of the change in a module.

3 Experimental methodology

In this section we describe the experimental methodology to obtain the information necessary for architecture–based analysis. In particular, the experimental methodology seeks to obtain:

- Architecture of the application as characterized by its intercomponent transition probabilities.
- Time–dependent coverage behavior of each component.

3.1 Selecting the application

The Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [37] 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. The choice of designating a file as a component was based on a tradeoff between the number of components, their size and the ease with which the necessary data can be collected. It was also due to the fact that each file hosts a group of related functions which would be placed in a single component, had the original design of SHARPE been component–based.

3.2 Executing 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 to execute the application. ATAC, which is a part of Telcordia Software Visualization and Analysis Tool Suite (TSVAT) [22] was used to measure coverage during the execution of

Table 1: Lines of code for components of SHARPE

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

the application with these test cases. Coverage measurements also enabled us to assess how well the code of SHARPE could be covered by this suite. The use of ATAC focuses on three main activities: instrumenting the software, executing software tests, and measuring coverage to determine how well the code has been covered. 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, test cases can be executed and 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. In our experiments, we used block coverage, since 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%. The individual coverage for each component is listed in Table 2.

Block–level coverage measurements obtained during the execution of the application are used to obtain the time–dependent coverage behavior for each component as described in Section 3.3. They are also aggregated to determine interactions among the functions, and subsequently among the files of the application as described in Section 3.4. We would like to note

[2]TSVAT can report coverage with respect to the function entry, block, decision, c–uses and p–uses criteria [23].
[3]This definition assumes that the underlying hardware does not fail during the execution of a block.

Table 2: Block coverage for components of SHARPE

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

that the proposed methodology uses block–level coverage measurements since they are readily available from the coverage analysis tool. However, block–level coverage measurements are not imperative to the use of this methodology. Since the block–level coverage information is subsequently condensed to obtain the time–dependent coverage behavior for the whole component, any profiling tool which reports coverage information on a per function basis would also provide necessary information. Similarly, since block–level coverage information is aggregated to determine the interactions between functions and files, any profiler which records similar execution information for individual functions could also provide the information necessary for determining the interactions. For applications developed using component–based software engineering paradigm, monitoring the components at their boundaries by recording the events originating from the components could also provide the necessary information to determine the architecture of the application. If the source code of the component is available then instrumenting the source code and collecting execution profiles will provide the coverage information necessary for determining coverage behavior. In the event that the source code is not available, then instrumentation of the object code could provide the necessary coverage data.

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 and the expected number of faults for each component (see Equation (6)). We first describe how to obtain the expected coverage behavior for each component:

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 as a function of time. To avoid any possible bias, the test ordering for each run was generated randomly.

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 parts of Table 3. The third part of Table 3 shows the expected block coverage, which is the average of the block coverage over the runs in the first two tables.

Since repeated execution of the application with a big test set is very costly, a tradeoff 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.

The expected number of faults in each component can be obtained using a variety of approaches. It may be estimated based on the failure data collected during unit testing of each component. It may also be estimated based on the software metrics of the component. We estimate the expected number of faults in each component using the fault density approach [30] (See Section 4 for details). We note that the fault density approach is a specific example of the broad class of approaches that are based on software metrics.

3.4 Determining the architecture of the application

The architecture of the application in terms of the intercomponent transition probabilities is determined based on 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

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

Execution ordering: t_1 followed by t_2

Block coverage (%)	Time (second)
0	0
0	1
50	2
50	3
50	4
60	5

Execution ordering: t_2 followed by t_1

Block coverage (%)	Time (second)
0	0
0	1
0	2
25	3
25	4
60	5

Expected block coverage

Expected block coverage (%)	Time (second)
0	0
0	1
25	2
37.5	3
37.5	4
60	5

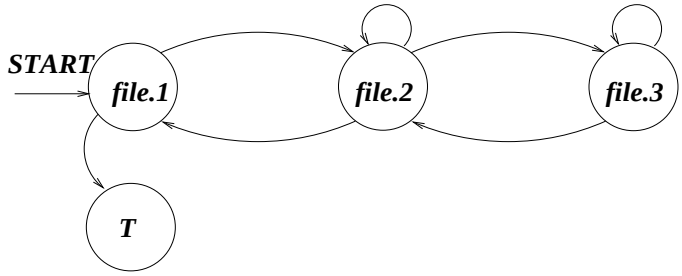

Figure 4: An example of file level transitions

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 SHARPE were computed based on the execution counts extracted from the ATAC trace files. The execution counts were 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.

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 $file.2$, of which the first block 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 4, where T indicates the terminating

[4]Each system function is treated as a single basic block.

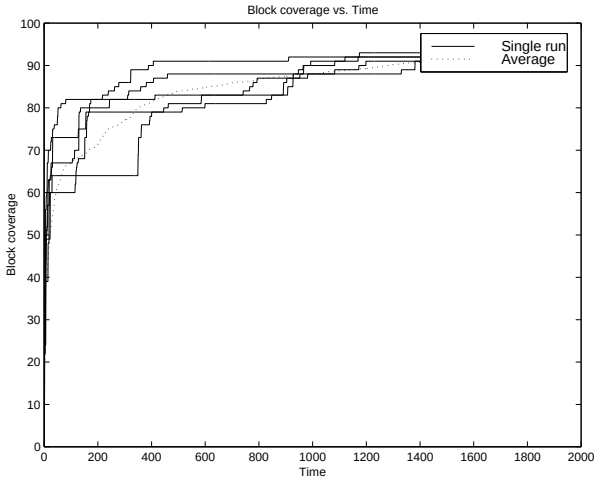

Figure 5: Coverage behavior as a function of time for $share.c$

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. As a result, $file.1$ 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 executed using the minimal test set as explained in Section 3.3 with 40 random orderings of the test cases. The coverage was measured for each file during 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 5 for $share.c$

Given measured coverage, to obtain the failure behavior of an individual component, we need an estimate of the expected number of faults a_i in the component. We obtain this estimate

[5] In our experiment, the application was SHARPE

using the fault density approach [20]. 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 [20]. The expected number of faults in component i, denoted by a_i is given by:

$$a_i = \frac{FD * l_i}{1000} \tag{15}$$

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 counts V_j as explained in Section 2.1. The execution counts for the file $bitlib.c$ are shown on the left hand side in Figure 6, and the corresponding branching probabilities are shown on the right side.

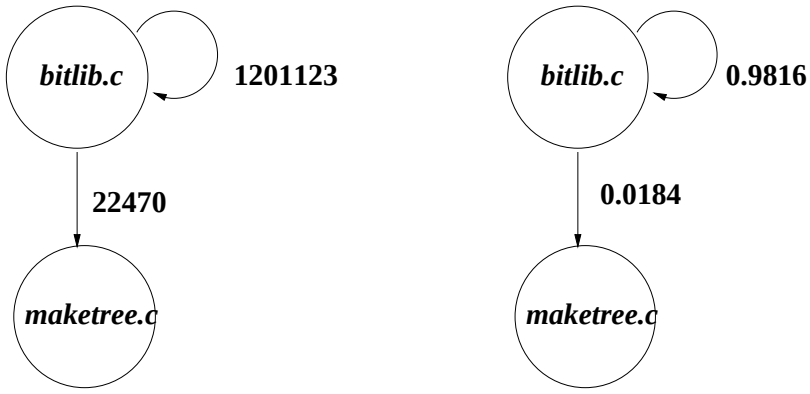

Figure 6: Execution counts and branching probabilities for $bitlib.c$

The branching probabilities are computed using per block execution information, so 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). In general, the execution time of different blocks of the same file might significantly differ. However, in our approach, a basic block is defined as a set of statements that are all executed together or none at all. According to this definition, a basic block constitutes on an average about 2 lines of C code. The execution time of a set of 2 lines of C code is not likely to differ significantly from one set to the other. As a result, in our set up it is reasonable to assume that the execution time of each block in a file is the same. Also, since the number of block executions is likely

Table 4: Expected time per component of SHARPE

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
analyze.c	2.3076e-06	inshare.c	1.9608e-07	maketree.c	0.00
multpath.c	0.00	results.c	2.1899e-06	in_qn_pn.c	1.9109e-07
share.c	1.3705e-05	cg.c	0.00	inchain.c	1.1777e-05
pfqn.c	0.0011	bind.c	2.6885e-05	bitlib.c	0.00
reachgraph.c	0.00	sor.c	7.6910e-06	newcg.c	0.00
mpfqn.c	3.6359e-07	phase.c	0.00	newphase.c	0.00
util.c	2.7770e-04	newlinear.c	8.3674e-08	cexpo.c	2.7635e-06
inspade.c	6.4805e-07	expo.c	7.3790e-07	read1.c	3.3561e-04
indist.c	1.8069e-07	symbol.c	0.00	ftree.c	6.1570e-04
debug.c	0.00	uniform.c	0.0	mtta.c	1.0705e-06

to be large, small variations in the execution times of each block are likely to average out. We compute the expected execution time of a block using the following equation:

$$EB = \frac{TT}{TB} \tag{16}$$

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 (16) is 0.0000057257. The expected time spent by the application in each component is summarized in Table 4, and the mean value function of the each component is summarized in Table 5.

The expected time to completion as computed from Equation (4) is 6.80 sec. The reliability estimate of the application from Equation (12) is 0.9903.

4.1 Validation

The performance estimate or the mean time to completion of the application obtained from our experiment was validated using the mean time to completion computed empirically. If the total time taken to execute R test cases is Q, the empirical mean time to completion ($ETTC$) can be computed using the following expression:

$$ETTC = \frac{Q}{R} \tag{17}$$

For our experiment Q was 5582 seconds and R, which is the number of test cases was 735 test cases. The mean time to completion computed empirically is thus 7.54 seconds. The expected time to completion using our approach (6.80 seconds), and the mean time to completion computed empirically are quite close. The small discrepancy between the empirical expected time to completion and the one computed using our approach could be due to one or more of the following reasons: 1) The test suite used in the study is not completely representative of actual usage of the application, and hence the branching probabilities computed based on the test suite do not represent the true branching probabilities. 2) 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.

The component with the maximum expected time is the "performance bottleneck". Table 4 indicates that the component $pfqn.c$ is the performance bottleneck.

The reliability estimate obtained using our approach may be validated by reinserting the faults discovered during integration testing and operational use of the application followed by testing the application based on its operational profile. Our present research involves the development of an operational profile for SHARPE followed by the validation of the reliability estimates. In the absence of validation, our approach can be used to determine the relative importance of individual components on the overall application reliability. The contribution of an individual component to the overall application reliability is a function of the utilization of the component as determined by its visit counts, the residual number of faults in the component, and the time–dependent coverage behavior of the component. We note that the utilization and time–dependent coverage behavior of each component is obtained from experimental data, whereas the residual number of number of faults in each component is estimated using the fault density approach. If the fault density for each component is assumed to be identical, then the relative criticality of each component will depend on its utilization and time–dependent coverage behavior. Since the utilization and time–dependent coverage behavior of each component is determined using the same experimental set up, the relative criticalities of the components computed using this approach are likely to hold, even if the absolute reliability values do not.

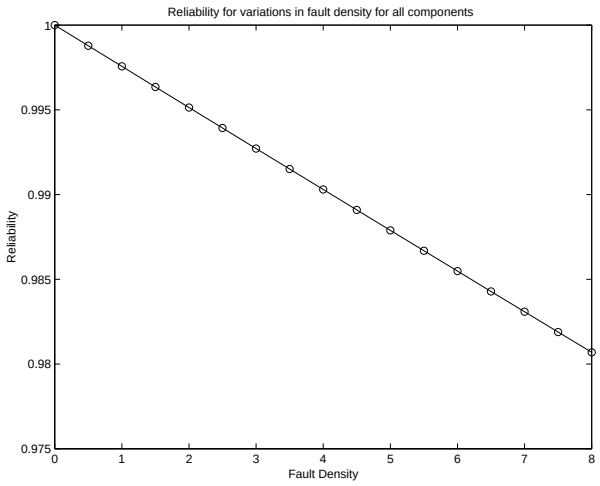

Figure 7: Effect of variations in fault density of the entire application

The relative priority of each component can be determined from their mean value functions listed in Table 5, where the component with the highest mean value function is most critical or is the "reliability bottleneck". From the table, it can be seen that the component $pfqn.c$ is the reliability bottleneck.

4.2 Sensitivity analysis

The approach presented in this paper allows us to capture the dependence between overall application performance (reliability), application architecture, and the performance (reliability) behavior of its individual components into a parameterized analytical model. This parameterized analytical model can be very valuable to conduct sensitivity analysis, or to analyze the impact of the variation in the performance (reliability) parameters for the individual components on the overall application performance (reliability). We demonstrate the ease of use of a parameterized analytical models for sensitivity analysis using the following two examples:

To determine the effect of the variation in fault density of the entire application on the reliability estimate obtained using our approach, we computed the reliability for various values of the fault density. Figure 7 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. This analysis can help us answer a question such as: what is the percentage increase in the reliability if the fault density of the application is reduced from 4.0 to 1.0?

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 have information regarding the failure behavior

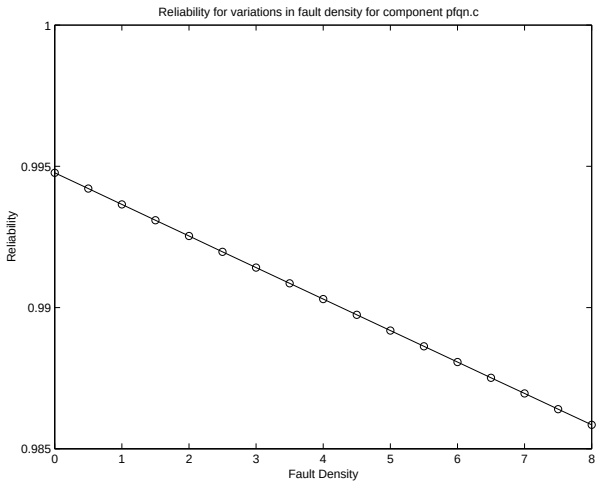

Figure 8: Effect of variations in fault density of $pfqn.c$

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. Without loss of generality, we assume that the component $pfqn.c$ was picked off the shelf. The reliability of the application for variations in the fault density of $pfqn.c$, is shown in Figure 8. As expected, the reliability decreases with increasing values of fault density of the component $pfqn.c$. Based on such analysis we can answer questions such as what is the percentage increase in application reliability if component A is procured from vendor X as opposed to vendor Y, sine the component from vendor X has a higher reliability than the component from vendor Y, but this higher reliability comes at an increased cost?

5 Conclusions and future research

In this paper we have described an experimental approach to extract the parameters of an analytical architecture–based software performance and reliability model based on the information obtained from the execution of an application. The experimental approach can be used for applications that are developed in a ground up manner and provide complete access to the source code. We use coverage measurements obtained during the execution of the application to extract parameters. We have demonstrated our experimental approach using the Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE). We have illustrated how a parameterized analytical model can be used for sensitivity analysis as well as to identify performance and reliability bottlenecks in an application.

Our future research includes designing and conducting experiments to validate the reliability predictions obtained using this approach. Experimental verification of the assumptions underlying various architecture–based software reliability models is also currently underway. Development of empirical methodologies to parameterize the architecture–based performance and reliability models of applications developed using the component–based software development paradigm is also a subject of future research.

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 TSVAT team at Telcordia Technologies, especially Saul London, for his invaluable help and guidance in the use of ATAC.

References

[1] "COM". *http://ww.microsoft.com/com*.

[2] "CORBA 3.0". *http://www.omg.org/news/pr98/component.html*.

[3] "Enterprise JavaBeans". *http://java.sun.com/products/ejb*.

[4] "JavaBeans". *http://www.sun.com/beans/spec.html*.

[5] M. Chen, M. R. Lyu, and E. Wong. "An empirical study of the correlation between code coverage and reliability estimation". In *Proc. of METRICS'96*, pages 133–141, Berlin, Germany, March 1996.

[6] M. H. Chen, M. R. Lyu, and W. E. Wong. "Effect of code coverage on software reliability measurement". *IEEE Trans. on Reliability*, 50(2):165–170, June 2001.

[7] R. C. Cheung. "A user–oriented software reliability model". *IEEE Trans. on Software Engineering*, SE-6(2):118–125, March 1980.

[8] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. "Integration testing using interface mutation". In *Proc. of Seventh Intl. Symposium on Software Reliability Engineering*, pages 112–121, White Plains, NY, October 1996.

[9] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. "Interface mutation: An approach for integration testing". *IEEE Trans. on Software Engineering*, 27(3):228–247, March 2001.

[10] W. W. Everett. "Software component reliability analysis". In *Proc. of Application Specific Software Engineering and Technology*, Dallas, TX, March 1999.

[11] 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.

[12] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini. "On the correleation between code coverage and software reliability". In *Proc. Sixth Intl. Symposium on Software Reliability Engineering*, pages 124–132, Tolouse, France, October 1995.

[13] S. Gokhale, M. R. Lyu, and K. S. Trivedi. "Reliability simulation of component-based software systems". In *Proc. of Ninth Intl. Symposium on Software Reliability Engineering (ISSRE 98)*, pages 192–201, Paderborn, Germany, November 1998.

[14] S. Gokhale, T. Philip, P. N. Marinos, and K. S. Trivedi. "Unification of finite failure non-homogeneous Poisson process models through test coverage". In *Proc. Intl. Sypmosium on Software Reliability Engineering (ISSRE 96)*, pages 289–299, White Plains, NY, October 1996.

[15] S. Gokhale and K. S. Trivedi. "Structure–based software reliability prediction". In *Proc. of Fifth Intl. Conference on Advanced Computing (ADCOMP 97)*, pages 447–452, Chennai, India, December 1997.

[16] K. Goseva-Popstojanova, A. P. Mathur, and K. S. Trivedi. "Comparison of architecture–based software reliability models". In *Proc. of Intl. Symposium on Software Reliability Engineering*, Hong Kong, November 2001.

[17] K. Goseva-Popstojanova and K. S. Trivedi. "Architecture–based approach to reliability assessment of software systems". *Performance Evaluation*, 45(2-3), June 2001.

[18] D. Hamlet. "Are we testing for true reliability ?". *IEEE Software*, 13(4):21–27, July 1992.

[19] D. Hamlet, D. Woit, and D. Mason. "Theory of software reliability based on components". In *Proc. of Intl. Conference on Software Engineering*, pages 361–370, Toronto, Canada, 2001.

[20] 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.

[21] M. Hiller, A. Jhumka, and N. Suri. "An approach for analysing the propagation of data errors in software". In *Proc. of Dependable Systems and Networks*, June 2001.

[22] 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.

[23] 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.

[24] 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.

[25] S. Krishnamurthy and A. P. Mathur. “On the estimation of reliability of a software system using reliabilities of its components”. In *Proc. of Eighth Intl. Symosium on Software Reliability Engineering*, pages 146–155, Albuquerque, New Mexico, November 1997.

[26] P. Kubat. “Assessing reliability of modular software”. *Operations Research Letters*, 8(35–41), 1989.

[27] J. C. Laprie and K. Kanoun. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter “Software Reliability and System Reliability”, pages 27–69. McGraw-Hill, New York, NY, 1996.

[28] J. Ledoux and G. Rubino. “A counting model for software reliability analysis”. *IASTED Journal on Simulation*, 1997.

[29] J. Ledoux and G. Rubino. “Simple formulas for counting processes in reliability models”. *Theory of Applied Probability*, 1997.

[30] M. Lipow. “Number of faults per line of code”. *IEEE Trans. on Software Engineering*, SE-8(4):437–439, July 1982.

[31] B. Littlewood. “A reliability model for Markov structured software”. In *Proc. 1975 Int’l Conf. Reliable Software*, pages 204–207, Los Angeles, CA, April 1975.

[32] B. Littlewood. “A semi–Markov model for software reliability with failure costs”. In *Proc. Symp. Comput. Software Engineering*, pages 281–300, Polytechnic Institute of New York, April 1976.

[33] B. Littlewood. “Software reliability model for modular program structure”. *IEEE Trans. on Reliability*, R-28(3), August 1979.

[34] Y. K. Malaiya. "The relationship between test coverage and reliability". Technical Report CS-94-110, Dept. of Computer Science, Colorado State University, Colorado, March 1994.

[35] J. D. Musa. "Operational profiles in software-reliability engineering". *IEEE Software*, 10(2):14–32, March 1993.

[36] D. L. Parnas, P. C. Clements, and D. M. Weiss. "The modular structure of complex systems". *IEEE Trans. on Software Engineering*, SE-11(3):259–266, March 1985.

[37] 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.

[38] K. Seigrist. "Reliability of systems with Markov transfer of control". *IEEE Trans. on Software Engineering*, 14(7):1049–1053, July 1988.

[39] K. Seigrist. "Reliability of systems with Markov transfer of control, II". *IEEE Trans. on Software Engineering*, 14(10):1478–1480, October 1988.

[40] M. L. Shooman. "Structural models for software reliability prediction". In *Proc. 2nd Int'l Conf. Software Engineering*, pages 268–280, San Fransisco, CA, October 1976.

[41] N. D. Singpurwalla and S. P. Wilson. *"Statistical Methods in Software Engineering: Reliability and Risk"*. Springer Verlag, New York, NY, 1999.

[42] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley, 2001.

[43] J. Voas, F. Charron, and L. Beltracchi. "Error propagation analysis studies in a nuclear research code". In *Proc. of 1998 IEEE Aerospace Conference*, Snowmass, CO, 1998.

[44] W. Wang, Y. Wu, and M. H. Chen. "An architecture-based software reliability model". In *Proc. of Pacific Rim Dependability Symposium*, Hong Kong, December 1999.

[45] 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.

[46] M. Xie and C. Wohlin. "An additive reliability model for the analysis of modular software failure data". In *Proc. Sixth Intl. Symposium on Software Reliability Engineering*, pages 188–193, Tolouse, France, October 1995.

[47] S. Yacoub, B. Cukic, and H. Ammar. “Scenario-based analysis of component-based software”. In *Proc. of Tenth Intl. Symposium on Software Reliability Engineering*, Boca Raton, FL, November 1999.