Comparison of Architecture-Based Software Reliability Models

Katerina Goševa–Popstojanova[1] *, Aditya P. Mathur[2], Kishor S. Trivedi[3]
[1]*Lane Dept. of Computer Sci. and Electrical Eng., West Virginia University, Morgantown, WV 26506*
[2]*Computer Sciences Department, Purdue University, West Lafayette, IN 47907*
[3]*Dept. of Electrical and Computer Eng., Duke University, Durham, NC 27708*

Abstract

Many architecture-based software reliability models have been proposed in the past without any attempt to establish a relationship among them. The aim of this paper is to fill this gap. First, the unifying structural properties of the models are exhibited and the theoretical relationship is established. Then, the estimates provided by the models are compared using an empirical case study. The program chosen for the case study consists of almost 10,000 lines of C code divided into several components. The faulty version of the program was obtained by reinserting the faults discovered during integration testing and operational usage and the correct version was used as an oracle. A set of test cases was generated randomly accordingly to the known operational profile. The results show that 1) all models give reasonably accurate estimations compared to the actual reliability and 2) faults present in the components influence both components reliabilities and the way components interact.

1. Introduction

Since the early 1970s a number of models have been proposed for estimating software reliability. The most widely known are the models that estimate reliability growth during testing phase [5], [6], [21]. These so called black–box models treat the software as a monolithic whole, without an attempt to model internal structure. Only interactions with external environment are considered, and usually no information other than the failure data is used.

With growing complexity and emphasis on reuse, the current software engineering practice emphasizes development of component based systems. The existing black–box models are clearly inappropriate to model such a large component–based software system. Instead, there is a need for a white–box approach which estimates system reliability taking into account the information about the architecture of the software made out of components. The motivation for the use of architecture–based software reliability approach includes the following:

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

Many architecture–based software reliability models have been proposed in the past, mostly by ad hoc methods. For extensive survey which proposes classification of the architecture–based software reliability models, contains a detailed description of the key models, and discussion on the underlying assumptions, usefulness and limitations the reader is referred to [9]. The aim of this paper is twofold: 1) to establish a theoretical relationship among different architecture–based software reliability models expanding our earlier work [8], and 2) to compare the estimates provided by the models based on an empirical case study [17].

The rest of the paper is organized as follows. The unifying structural properties of the models are exhibited in Section 2. The key models are described in detail and their theoretical relationship is established in Section 3. The comparison of the models based on an empirical case study is presented in Section 4. The concluding remarks are presented in Section 5.

2. Common structural properties

2.1. System decomposition

Although there is no universally accepted definition, a component is conceived as a logically independent unit of the system which performs a well–defined function. This implies that a component can be designed, implemented, and tested independently. Component definition is a user level task that depends on the factors such as system being analyzed, possibility of getting the required data, etc.

*This work was done while the author was with Duke University

There is a trade off in defining the components. Too many small components may pose difficulties in measurements, parametrization, and solution of the model. On the other hand, too few components may cause the distinction of how different components contribute to the system failures to be lost. The level of decomposition clearly depends on the tradeoff between the number of components, their complexity and the available information about each component.

2.2. Software architecture

Software behavior with respect to the manner in which different components interact is defined through the software architecture. Interaction occurs only by transfer of execution control. In the case of sequential software, at each instant, control lies in one and only one of the components. The architecture of an application may not always be readily available. In such cases, it has to be extracted from the source code or the object code of the application. Techniques and tools for extraction of architectural information are described in our earlier work [9].

In architecture–based approach, one must model the interaction of all components. In well designed system, interaction among components is limited. During the early phases of software life cycle, each component could be examined to find with which components it cannot interact. If control can flow between two components it can be described by non–zero transition probabilities that may be available by analyzing program structure and using known operational profiles [20], [27]. During the design phase, before actual development and integration phases, the transition probabilities can be estimated by simulation. As new data become available during the integration phase the estimates have to be updated [7].

The estimation of transition probabilities is affected by the user's operational profile. Upgrades to software might invalidate any existing estimate of operational profile because new features can change the ways in which the software is used. Therefore, it will be necessary to revise the architecture that describes component interaction and modify transition probabilities.

2.3. Failure behavior

In the next step, the failure behavior is defined and associated with the software architecture. Failure can happen during an execution period of any component or during the control transfer between two components. The failure behavior of the components and of the interfaces between components can be specified in terms of their reliabilities or failure rates.

Assessing the reliability of software components clearly depends on the factors such as whether or not component code is available, how well the component has been tested, and whether it is a reused or a new component. Several techniques for estimating component's reliability have been proposed. Software reliability growth models can be applied to each software component exploiting component's failure data obtained during testing [4], [7], [10]. However, due to the scarcity of failure data it is not always possible to use software reliability growth models. Another possibility is to estimate component's reliability from explicit consideration of non–failed executions, possibly together with failures [18], [19]. In this context, testing is not an activity for discovering faults, but an independent validation activity. The problem which arises with these models is the large number of executions necessary to establish a reasonable statistical confidence in the reliability estimate. Finally, one can use fault injection technique to estimate component's reliability. However, fault–based techniques are only as powerful as the range of fault classes that they simulate. In the context of component's reliability estimation more research is needed when it comes to the certification of COTS [24] and reused software components [25].

There is little information available about interface failures, apart from general agreement that they exist separately from component failures which are revealed during the unit testing. When an interface consists of items such as global variables, parameters, and files, it is not clear how to estimate its reliability. Some explanation and analysis about the interfaces between components has been presented in [23]. Also, method for integration testing proposed in [3] seems promising for estimating interface reliabilities.

2.4. Combining software architecture with failure behavior

Depending on the method used to combine the architecture of the software with the failure behavior, three essentially different approaches can be recognized [9]:

State-based models use the control flow graph to represent software architecture and estimate software reliability analytically. They assume that the transfer of control between components has a Markov property, that is, model software architecture with a discrete time Markov chain (DTMC), continuous time Markov chain (CTMC), or semi Markov process (SMP). These can be further classified into absorbing and irreducible. The former represents applications that operate on demand, while the later is well suited for continuously operating software applications. Accordingly to the solution method, state–based models can be classified as either composite or hierarchical. The composite method combines the architecture of the software with the failure behavior into a composite model which is then solved to predict reliability of the application. The other possibility is to take the hierarchical approach, that is, to solve first the architectural model and then to superimpose the failure behavior on the solution of the architectural model in order to predict reliability.

Path–based models compute software reliability considering the possible execution paths of the program. A sequence of components along different paths is obtained either experimentally by testing or algorithmically. The reliability of each path is computed by multiplying the reliabilities of the components along that path. Then, the system reliability is estimated by averaging path reliabilities over all paths.

Additive models assume that each component reliability can be modeled by non–homogeneous Poisson process (NHPP). Then, system failure process is also NHPP with the cumulative number of failures and failure intensity function that are the sums of the corresponding functions for each component. Since additive models [4], [26] do not explicitly consider software architecture they are not considered further in this paper.

In the following section the theoretical relationship among existing architecture–based software reliability models is established. We describe in detail models that are suitable for comparison based on the empirical data obtained from the case study.

3. Theoretical relationship

3.1. State–based models of terminating applications

First, consider the state–based models of a terminating software application. These models assume that a control flow graph has a single entry and a single exit node representing components at which execution begins and is terminated, respectively. Note that this is not a fundamental requirement; the models can easily be extended to cover multientry, multiexit graphs. The relevant measure for these models is the reliability R of the single execution of software application.

Cheung model [2] is one of the earliest models that considers software reliability with respect to the components utilization and their reliabilities. The transfer of control among components is described by an absorbing DTMC with a transition probability matrix $P = [p_{ij}]$, where $p_{ij} = Pr\{\text{program transits from component } i \text{ to component } j\}$. Components fail independently and the reliability of the component i is the probability R_i that the component performs its function correctly.

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

$$R = S(1,n)\, R_n. \tag{1}$$

Basically, the reliability is equivalent to the sum of reliabilities of all paths that start at the entry node and end at the exit node C, including the possibility of infinite number of paths due to the loops that might exist between two or more components.

The model proposed by *Kubat* [12] includes the information about execution time of each component, thus resulting in an SMP as a model of software architecture. Transitions between components follow a DTMC with initial state probability vector $q = [q_i]$ and transition probability matrix $P = [p_{ij}]$. The sojourn time during a visit in component i has the pdf $g_i(t)$. When component i is executed, failures occur with constant failure intensity λ_i.

The solution method taken in this work is hierarchical. The reliability of component i is estimated as the probability that no failure occurs during its execution

$$R_i = \int_0^{\infty} e^{-\lambda_i t} g_i(t)\, dt. \tag{2}$$

During a single execution of the software application each component i is executed a random number of times, denoted by N_i. Thus $R_i^{N_i}$ can be considered as the equivalent reliability of component i that takes into account its utilization. Assuming that components fail independently the system reliability becomes $\prod_{i=1}^{n} R_i^{N_i}$. Based on Taylor's series expansion it is shown that the first order approximation of $E[\prod_{i=1}^{n} R_i^{N_i}]$ can be used for the system reliability

$$R \approx \prod_{i=1}^{n} R_i^{V_i} \tag{3}$$

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

$$V_i = q_i + \sum_{j=1}^{n} V_j\, p_{ji}. \tag{4}$$

Note that once component reliabilities are estimated using equation (2) the solution approach reduces to the hierarchical treatment of the Cheung model [2].

The hierarchical model proposed by *Gokhale et al.* [7] differs in the approach taken to estimate the component reliabilities; it considers time dependent failure rates $\lambda_i(t)$ and the utilization of the components through the cumulative expected time spent in the component per execution $V_i t_i$, where t_i is the expected time spent in a component i per visit. The special case of this model that assumes constant failure rates λ_i is equivalent to the special case of Kubat model [12] that assumes deterministic execution times $g_i(t) = t_i$.

Above software reliability models assume that components fail independently and that a component failure will ultimately lead to a system failure. In hardware system reliability it is generally considered that all components are continuously active which corresponds to the usual equation for the reliability of a series system $R = \prod_{i=1}^{n} R_i$. The key question in software reliability is how to account for component's utilization, that is, for the stress imposed on each component during execution. In the case of the hierarchical models [7], [12] the expected number of times each component is executed V_i is taken as a measure of component's utilization.

3.2. State–based models of continuously running applications

Now, consider the models of continuously running applications. The relevant measures for these models are waiting time to first failure, or equivalent system failure rate. *Littlewood model* [15] is the earliest architecture–based software reliability model. It describes software architecture with an irreducible CTMC and considers both component and interface failures. *Laprie model* [13] which considers only component failures is a special case of [15]. An extension of [13] and [15] that considers the way failure processes affect the execution and deals with the delays in recovering an operational state was proposed recently by *Ledoux* [14]. Another generalization of [15] proposed in [16] describes software architecture of continuously running application by an irreducible SMP assuming that the time spent in each component has a general distribution.

For the sake of comparison we discuss in detail *Laprie model* [13]. This model describes a software system made up of n components by a CTMC. The parameters are the mean execution time of a component i given by $\bar{t_i} = 1/\mu_i$ and the probability p_{ij} that component j is executed after component i given that no failure occurred during the execution of component i. It is assumed that each component fails with constant failure rate λ_i.

The model of the system is an $n+1$ state CTMC where the system is up in the states $i, 0 \leq i \leq n$ and the $(n+1)$th state being the down state reached after a failure occurrence. The associated generator matrix between the up states $B = [b_{ij}]$ is defined by $b_{ii} = -(\mu_i + \lambda_i)$ and $b_{ij} = p_{ij}\mu_i$, for $i \neq j$. It can be seen as the sum of two generator matrices: B' that governs the execution process, with diagonal entries equal to $-\mu_i$ and off-diagonal entries to $p_{ij}\mu_i$, and B'' that governs the failure process, with diagonal entries equal to $-\lambda_i$ and off-diagonal entries to zero. The assumption that the failure rates are much smaller than the execution rates $\lambda_i \ll \mu_i$ leads to the asymptotic behavior relative to the execution process which enables to adopt the hierarchical solution method. As a consequence, the system failure rate tends to

$$\lambda_S = \sum_{i=1}^{n} \pi_i \lambda_i \tag{5}$$

where the steady state probability vector $\pi = [\pi_i]$ is the solution of $\pi B' = 0$.

The above result deserves a few comments. First, the asymptotic Poisson process can be seen as a superposition of Poisson processes with parameters $\pi_i \lambda_i$, which is closely related to the approach taken in the additive models. Further, consider the probability that there will be no failure up to time t, that is, the system reliability

$$R(t) \approx e^{-\lambda_S\, t} = e^{-\sum_{i=1}^{n} \pi_i \lambda_i\, t} = \prod_{i=1}^{n} e^{-\lambda_i\, \pi_i t}. \tag{6}$$

π_i represents the proportion of time spent in state i in the absence of failure; thus $\pi_i t$ represents the cumulative execution time spent in a component i up to time t. For hardware systems it is considered that all components are continuously active which corresponds to making all the π_i's equal to 1. From the reliability point of view, this leads to the usual equation for a series system with a number of subsystems with exponentially distributed times to failure.

Next we establish the relation of this model with the models of terminating applications. CTMC that represents Laprie model [13] is irreducible since it considers the continuity of execution and it is assumed that after a failure the execution process is restarted instantaneously. Any long random realization of the irreducible CTMC can be partitioned to single software runs that are equivalent to realizations from the fixed starting state to recurrence of starting state. For simplicity we assume that CTMC has a single starting state and a single exit state that represents the final task before the next execution. Having in mind that $R_i = \int_0^\infty e^{-\lambda_i t} \mu_i e^{-\mu_i t}\, dt = \frac{\mu_i}{\mu_i + \lambda_i}$, it can be shown that the embedded DTMC of this $(n+1)$ state CTMC is equivalent to the DTMC that represents Cheung model [2] with additional transitions (with probability 1) from both exit state and failure state to starting state which represent immediate reset/restart of program executions.

To compare the hierarchical solution with the models of terminating applications we need to partition the realization of the irreducible CTMC with generator matrix B' to single software runs that are equivalent to realizations from a starting state to recurrence of starting state. The execution time of a single run is a random variable with expectation equal to

$$\bar{t} = \text{E[time of one cycle]} = \sum_{i=1}^{n} V_i/\mu_i. \tag{7}$$

Also, it is well known that the proportion of time that irreducible CTMC spends in state i is given by

$$\pi_i = \frac{\text{E[time in state } i \text{ during one cycle]}}{\text{E[time of one cycle]}} = \frac{V_i/\mu_i}{\sum_{i=1}^{n} V_i/\mu_i}. \tag{8}$$

Using (6) for the first order approximation of $E[R(t)]$ we get

$$R(\bar{t}) \approx \prod_{i=1}^{n} \left(e^{-\lambda_i/\mu_i}\right)^{V_i} \tag{9}$$

where the term in the parentheses is the first order approximation for component reliabilities $R_i(\bar{t}_i) \approx e^{-\lambda_i \bar{t}_i} = e^{-\lambda_i/\mu_i}$. Clearly, this approximate solution for the reliability of a single software run is equivalent to the hierarchical solution given by (3).

3.3. Path-based models

Similar to state-based models, path-based models consider software architecture explicitly and assume that components fail independently. However, the method of combining software architecture with failure behavior is not analytical. Path-based models consider different paths that can be taken during software execution. They account for each component utilization along each path, as well as among different paths. One of the earliest architecture-based reliability models that introduces path-based approach was proposed by *Shooman* [22]. This model assumes that software execution can take fixed number of different paths. The frequency of occurrence of each path and its failure probability are assumed to be known.

The path-based model proposed by *Krishnamurthy and Mathur* [11] takes an experimental approach to obtain the path reliability estimates. Specifically, sequences of components along different paths are observed using the component traces collected during the testing. The component trace of a program P for a given test case tc, denoted by $M(P, tc)$, is the sequence of components m executed when P is executed against tc. A sequence of components along a path traversed for test case tc is considered as a series system, and assuming that components fail independently of each other it follows that the path reliability is

$$R_{tc} = \prod_{\forall m \in M(P,tc)} R_m. \tag{10}$$

The reliability estimate of a program with respect to a test set TS is obtained by averaging path reliabilities

$$R = \frac{\sum_{\forall tc \in TS} R_{tc}}{|TS|}. \tag{11}$$

Yacoub, Cukic and Ammar model [27] takes an algorithmic approach to estimate path reliabilities. A probabilistic model named Component Dependency Graph is constructed and the algorithm expands all branches starting from the entry node. The breadth expansions of the tree represent logical "OR" paths and are hence translated as the summation of reliabilities weighted by the corresponding probabilities of occurrence for each path. The depth of each path represents the sequential execution of components, the logical "AND", and is hence translated to multiplication of reliabilities. The depth expansion of a path terminates when the next node is an exit node (a natural end of an application execution) or when the summation of execution time of that thread sums to the average execution time. The later condition guarantees that the loops between two or more components do not lead to a deadlock.

The difference between the state-based and path-based approaches becomes evident when the control flow graph of the application contains loops. State-based models analytically account for the infinite number of paths due to the loops that might exist. In the case of path-based models either the number of paths is restricted to ones observed experimentally during the testing [11] or the depth traversal of each path is terminated using the average execution time of the application [27]. However, the difference will not be significant since long paths have low path probabilities. Note that path-based models are related to path testing which uses program's control flow graph as a structural model. Regarding the path selection criteria during path testing, it is suggested that it is better to take many short, simple paths than a few long, complicated paths [1].

4. Experimental comparison

4.1. Description of the case study

The program chosen for the case study provides language-oriented user interface which allows the user to describe the configuration of an array of antennas. Its purpose is to prepare a data file in accordance with a predefined format and characteristics from a user, given the array antenna configuration described using an appropriate Array Definition Language. The program was developed for the European Space Agency in C language and consists of almost 10,000 lines of code. It is divided into three subsystems: the Parser subsystem, the Computational subsystem, and the Formatting subsystem. This program has been used as a case study for investigating sensitivity of software reliability growth models to operational profile errors [20].

The choice of this program as a case study for experimental comparison of the models was based on the following [20]:

- The program is real and of typical size for this kind of application
- The programming language is widely used
- Faults reinserted are the real faults discovered during integration testing and operational use of the program
- A set of test cases is generated randomly accordingly to the known operational profile determined by interviewing the users of the program
- The program has been extensively used after the last fault removal without having new failures. This gold version of the program is used as an oracle during the experiment.

Analyzing the code it was possible to gather software architecture as shown in Figure 1 [17]. Components 1, 2, and 3 correspond to the Parser, Computational, and Formatting subsystems respectively. State E represents the completion of execution. The choice for the decomposition was made in order to reach a tradeoff between number of components, their size, and the ability to collect data needed for use in the models. More complex model would impose difficulties on the measurement task, especially in the operational phase when failures occur in principle less frequently. Similar situation can be found in the experiment presented in [10] which considers a telephone switching software system decomposed into only four components.

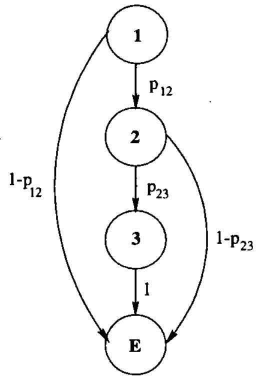

Figure 1. Program architecture

Faulty versions of the program were obtained by reinserting the faults discovered during integration testing and operational usage. In particular, two faults were reinserted in each of the components 1 and 2. Two faulty versions of the program were constructed; faulty version A consists of fault–free component 3 and faulty components 1 and 2, while faulty version B consists of fault–free components 1 and 3 and faulty component 2.

4.2. Model application

In this subsection we illustrate the application of architecture–based software reliability models on our case

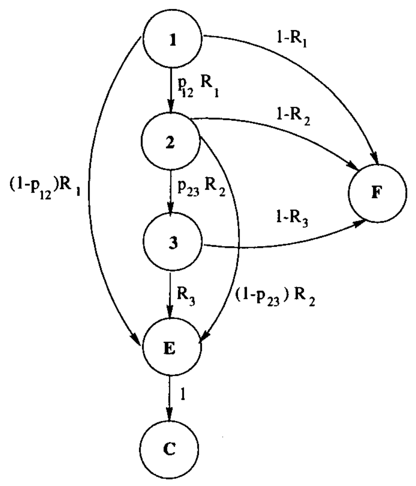

Figure 2. Composite model of a terminating application

study. DTMC presented in Figure 2 is a composite state-based model of a terminating application [2]. Using (1) it can be shown that the system reliability is given by

$$R = (1-p_{12})R_1 + p_{12}(1-p_{23})R_1R_2 + p_{12}p_{23}R_1R_2R_3. \tag{12}$$

If we take a hierarchical approach then we need to estimate the expected number of times each component is executed during a single execution of a software for a DTMC presented in Figure 1. Using (4) we get

$$V_1 = 1, \quad V_2 = p_{12}, \quad V_3 = p_{12}p_{23}. \tag{13}$$

The first order approximation for the system reliability given by (3) then becomes

$$R \approx R_1 R_2^{p_{12}} R_3^{p_{12}p_{23}}. \tag{14}$$

CTMC that represents continuously running application is presented in Figure 3. It is assumed that both successful termination E and failure F cause an immediate reset/restart to the initial state 1, denoted with dashed lines.

As we have already shown in the previous section embedded DTMC of this irreducible CTMC is equivalent to DTMC that represents Cheung model [2] with additional transitions (with probability 1) from E and F states to starting state 1. Subsequent visits to state 1 partition the realization into subsequences equivalent to single software runs. The reliability of a single software run with expected execution time $\bar{t}$ obtained by the approximate hierarchical solution of CTMC is equivalent to (14).

Component traces obtained during testing were used for estimating both transition probabilities for state–based models and path reliabilities for the experimental path-based model [11]. Since the software architecture does not contain loops it is obvious that the reliability obtained using

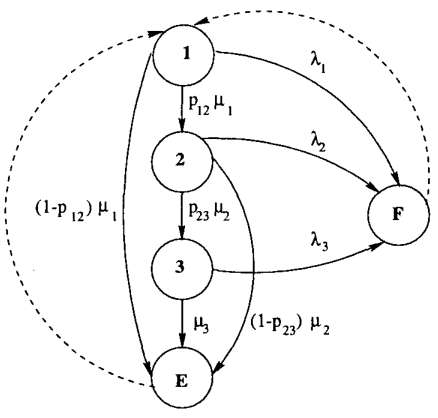

Figure 3. Composite model of a continuously running application

path–based model [11] will be the same as the reliability obtained using the composite state–based model [2]. For a simple software architecture given in Figure 1 it is easy to see that there are only three possible paths from component 1 to component E. Their probabilities of occurrence are given by $(1 - p_{12})$, $p_{12}(1 - p_{23})$, and $p_{12}p_{23}$, while the corresponding reliabilities are R_1, R_1R_2, and $R_1R_2R_3$ respectively. The system reliability obtained using the algorithmic path–based model [27] is a sum of paths reliabilities weighted by the corresponding probabilities of occurrence for each path which leads to the same equation as (12).

4.3. Parameter estimation and comparison of the results

In this subsection we compare the estimates provided by the models using empirical data from the case study[1]. Each faulty version of the program and the oracle were executed on the same test cases generated randomly on the basis of the operational profile. If their outputs disagree it is necessary to determine the component that has failed. Identification of the fault responsible for the failure is only aimed at determining which component has failed. Faults have not been removed and the number of failures includes recurrences due to the same fault. The reliability of each component was estimated using [19]

$$R_i = 1 - \lim_{n_i \to \infty} \frac{f_i}{n_i} \qquad (15)$$

where f_i is the number of failures of component i and n_i is the number of executions of component i in N randomly generated test cases. Component reliabilities estimated using (15) are given in Table 1. Since component 3 has not failed during integration testing and operational use of the program, there were no faults injected in this components and it is assumed that $R_3 = 1$.

[1]The number of significant figures is not intended to imply any accuracy in the estimates, but to illustrate the models.

R_1	R_2
0.8428	0.8346

Table 1. Components reliabilities

Transition probabilities were estimated using data obtained from the component traces collected during testing. Thus,

$$p_{ij} = \frac{n_{ij}}{n_i} \qquad (16)$$

where n_{ij} is the number of times control was transferred from component i to component j and $n_i = \sum_j n_{ij}$ is the number of times control reaches component i. Transition probabilities for both faulty versions are given in Table 2.

Version	p_{12}	p_{23}
A	0.5933	0.7704
B	0.7364	0.6866

Table 2. Transition probabilities

UNIX utility clock was used to measure the time spent in components for the gold version of the program (without faults injected). This function measures the time spent in a component only if it is at least 100 *ms*. Since in this program no component has execution time longer than 100 *ms* the mean execution times presented in Table 3 were measured using multiple repetitions. This means that it was not possible to test the hypothesis that components execution times are exponentially distributed. Components failure rates λ_i obtained using $R_i = \frac{\mu_i}{\mu_i + \lambda_i}$ are one order of magnitude smaller than execution rates μ_i which justifies the hierarchical approach taken in [13].

$1/\mu_1$	$1/\mu_2$	$1/\mu_3$
20 *ms*	6.5 *ms*	76 *ms*

Table 3. Mean execution times of components

We estimate the actual reliability of the software as

$$R = 1 - \lim_{N \to \infty} \frac{F}{N} \qquad (17)$$

where F is the number of system failures in N test cases generated randomly based on the operational profile. Table 4 compares the actual reliability with the reliability estimations provided by the composite model (terminating application [2] and single software run of continuously running application [13]), hierarchical model (terminating application [12] and single software run of continuously running application [13]), and path–based models (experimental approach [11] and algorithmic approach [27]).

In general, results show that all the models give reasonably accurate estimations compared to the actual reliability for each of the faulty versions. As expected, faulty version B which contains faults only in component 2 is more reliable than faulty version A. The results also show a strong

Faulty Version	Actual Reliability	Composite		Hierarchical		Path–based	
		Reliability	Error	Reliability	Error	Reliability	Error
A	0.7393	0.7601	2.81%	0.7571	2.41%	0.7601	2.81%
B	0.8782	0.8782	0%	0.8753	0.33%	0.8782	0%

Table 4. Comparison of the results

relation between the faults present in the components and the way components interact. Thus, as it can be seen from Table 2 transition probabilities p_{12} and p_{23} are heavily affected by the faults in component 1. It follows that the removal of two faults from component 1 in faulty version A which leads to faulty version B affects the expected number of times each component is executed during a single software run (see Table 5). This implies that even if the operational profile can be estimated accurately, the control flow, that is, components utilization might be significantly affected by some of the faults.

Version	V_1	V_2	V_3
A	1	0.5933	0.4571
B	1	0.7364	0.5056

Table 5. Expected number of component's visits

Since our case study indicates that fault removal affects the components reliabilities and the intra-component transition probabilities, in Figure 4 we illustrate how system reliability R varies as a function of R_1 and p_{12}, assuming a fixed value of $R_2 = 0.8346$[2]. The ranges chosen for p_{12} and R_1 are $[0.5, 0.8]$ and $[0.8, 1]$, respectively. The system reliability then ranges from 0.6941 to 0.9173.

The above relation between faults in the components and the way components interact has not been emphasised in previous research studies on architecture–based software reliability. This observation raises some interesting questions related to the sensitivity studies which usually assume fixed values for transition probabilities. The influence of the fault removal process on the components interaction and the implications on the way sensitivity studies are conducted need to be examined further in future experimental studies.

Finally, we emphasize that the estimates obtained using various models under the assumption that failure processes associated across different components are mutually independent are close to the actual reliability. However, there might be cases where this assumption is too strong, such as for example when most paths executed have components within loops and these loops are traversed sufficiently large number of times. Then, individual path reliabilities are likely to become low resulting in system reliability estimate much below its true reliability. In path–based model this issue can be resolved by collapsing multiple occurrences of a component on an execution path into k occurrences, where k is referred as the degree of independence [11]. In the case of state–based models it is possible to consider Markov chain of higher order [9].

[2]Although faults in component 1 influence both p_{12} and p_{23}, system reliability is not affected by p_{23} since it is assumed that $R_3 = 1$.

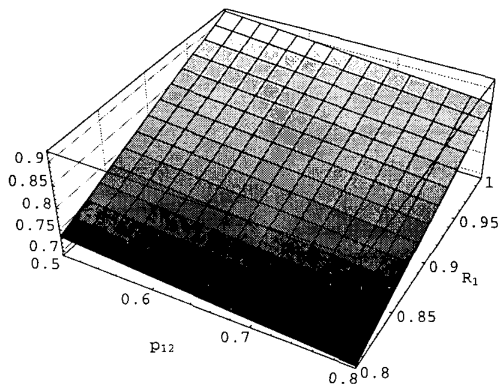

Figure 4. R as a function of p_{12} and R_1

4.4. An example of software architecture with loops

Next we consider a hypothetical example of a software architecture as in Figure 5 which has an additional transition from component 2 to component 1. This example is meant to illustrate how the components executed within a loop affect application reliability.

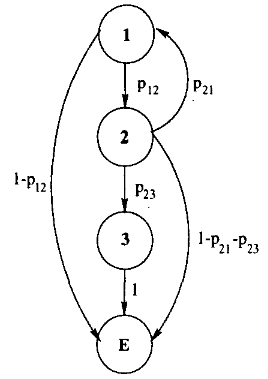

Figure 5. Program architecture with a loop

For the example in Figure 5 the application reliability obtained using the composite model (1) is given by

$$R = \frac{(1-p_{12})R_1 + p_{12}(1-p_{21}-p_{23})R_1R_2 + p_{12}p_{23}R_1R_2R_3}{1 - p_{12}p_{21}R_1R_2}. \quad (18)$$

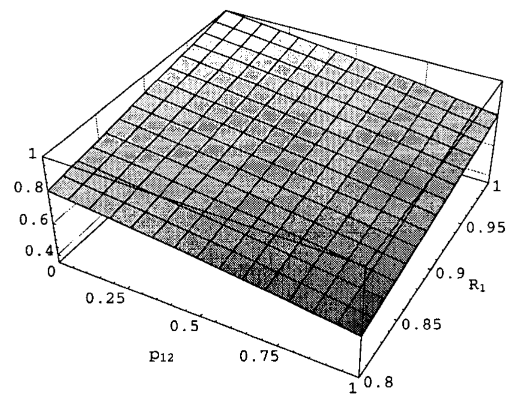

Figure 6. R as a function of p_{12} and R_1 for $p_{21} = 0.25$

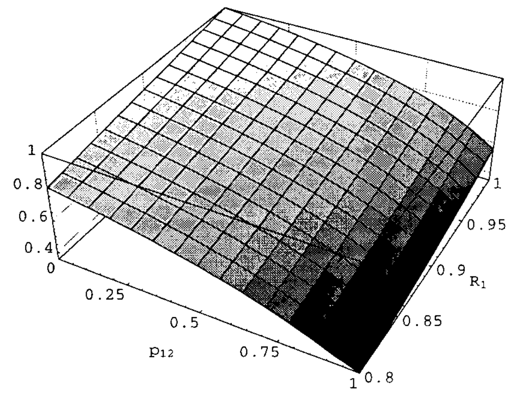

Figure 7. R as a function of p_{12} and R_1 for $p_{21} = 0.75$

Using this formula we plot in Figures 6 and 7 the variation in application reliability R as a function of p_{12} and R_1 for different values of p_{21}; the values for R_2, R_3, and p_{23} for this plot are fixed, with $R_2 = 0.8346$, $R_3 = 1$, and $p_{23} = 0.25$. Note that unlike the example without loops (see Figure 4), application reliability as a function of either p_{12} (with R_1 fixed) or R_1 (with p_{12} fixed) is non-linear. As indicated by these figures, higher values of transition probability p_{21} result in smaller application reliability due to the large number of times components 1 and 2 are executed within a loop.

The expected number of times that components are executed during a single execution of the software application obtained using (4) are given by

$$V_1 = \frac{1}{1 - p_{12}p_{21}}, \quad V_2 = p_{12}V_1, \quad V_3 = p_{23}V_2. \qquad (19)$$

As explained earlier the values of V_i are the clear indication of the component usage. It can be seen from Table 6 that V_1 and V_2 are significantly higher for higher values of transition probability p_{21}.

p_{21}	V_1	V_2	V_3
0	1	0.8	0.2
0.25	1.25	1	0.25
0.5	1.67	1.33	0.33
0.75	2.5	2	0.5

Table 6. Expected number of component's visits: $p_{12} = 0.8$ and $p_{23} = 0.25$

In our hypothetical example the number of paths is infinite due to the existence of a loop. If we consider all possible paths the application reliability is

$$\begin{aligned} R &= [(1-p_{12})R_1 + p_{12}(1-p_{21}-p_{23})R_1R_2 \\ &\quad + p_{12}p_{23}R_1R_2R_3] \qquad (20) \\ &\quad \cdot [1 + p_{12}p_{21}R_1R_2 + (p_{12}p_{21}R_1R_2)^2 + \cdots] \end{aligned}$$

which leads to the same solution as in the case of the composite model (18). Path–based approach [11] restricts the number of paths to ones observed experimentally. For the purpose of a comparative evaluation, we consider samples of test cases for the path–based approach that result in the same values of the transition probabilities as the ones used for the state–based models. Table 7 compares the reliability estimations provided by the composite model [2], hierarchical model [12], and path–based model [11]. In view of Table 7 the following observations are made. The error of hierarchical solution compared to the exact solution from the composite model increases for higher values of transition probability p_{21}. This is due to the fact that the values of variances of the number of times each component is executed increase with p_{21}. Further, the path–based approach results in estimates close to the composite state–based approach even with a small number of paths that we have considered. Finally, as anticipated by the remarks in previous subsection considering intra-component dependency by collapsing multiple occurrences of a component into a single occurrence ($k = 1$) leads to reliability estimates much higher than in the independent case. Although the observations reported above can not be validated with respect to an actual reliability, they do help us to outline the issues that need the attention in future experimental studies.

p_{21}	Composite	Hierarchical	Path–based independent	Path–based dependent
0	0.7313	0.7293	0.7623	0.7623
0.25	0.6873	0.6740	0.6809	0.7208
0.5	0.6261	0.5909	0.6039	0.7266
0.75	0.5351	0.4542	0.5256	0.7383

Table 7. Comparison of the results: $R_1 = 0.8428$, $R_2 = 0.8346$, $p_{12} = 0.8$, and $p_{23} = 0.25$

5. Conclusion

The paper establishes a clear theoretical relationship among architecture–based software reliability models that have been proposed in the past mostly by ad hoc methods. Then, it illustrates the models application on the case study and compares their estimations based on the empirical data. Results show that the estimations obtained by all models fit reasonably well with the actual reliability. Results also show a strong relation between faults present in the components and the way components interact which has not been emphasised in previous studies on architecture–based software reliability. This relation and its implications on the way sensitivity studies should be conducted are the subjects of our future research. In principle, a single case study cannot give considerable confidence that the same results will apply in different software systems. In order to validate the results of this paper, architecture–based software reliability models need to be applied to other software systems, preferably with more complex architecture.

Acknowledgements

We would like to thank Alberto Pasquini for providing the program used for the experiment. We also thank Pietro Michielen and Manuela Schiona for sharing with us results of their earlier work on the comparison of various models for reliability estimation using information regarding the architecture of the application. We acknowledge the financial support of NSF under the grant EEC-9714965. We also acknowledge the financial support of the Department of Defense, Alcatel, Stratus, and Telcordia by means of a CACC Core project, and Jet Propulsion Laboratory under the award JPL REE 1216658.

References

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, 1983.

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

[3] M. Delamaro, J. Maldonado, and A. P. Mathur. Integration testing using interface mutations. In *Proc. 7th Int'l Symp. on Software Reliability Engineering*, pages 112-121, 1996.

[4] W. Everett. Software component reliability analysis. In *Proc. Symp. Application–Specific Systems and Software Engineering Technology*, pages 204-211, 1999.

[5] W. Farr. Software reliability modeling survey. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 71-117. McGraw–Hill, 1996.

[6] A. L. Goel. Software reliability models: assumptions, limitations, and applicability. *IEEE Trans. on Software Engineering*, 11(12):1411-1423, 1985.

[7] S. Gokhale, W. E. Wong, K. Trivedi, and J. R. Horgan. An analytical approach to architecture based software reliability prediction. In *Proc. 3rd Int'l Computer Performance & Dependability Symp.*, pages 13-22, 1998.

[8] K. Goševa–Popstojanova, K. Trivedi, and A. P. Mathur. How different architecture based software reliability models are related? In *Fast Abstracts 11th Int'l Symp. on Software Reliability Engineering*, pages 25-26, 2000.

[9] K. Goševa–Popstojanova and K. Trivedi. Architecture–based approach to reliability assessment of software systems. *Performance Evaluation*, 45:179-204, 2001.

[10] K. Kanoun and T. Sabourin. Software dependability of a telephone switching system. In *Proc. 17th Int'l Symp. on Fault–Tolerant Computing*, pages 236-241, 1987.

[11] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Proc. 8th Int'l Symp. Software Reliability Engineering*, pages 146-155, 1997.

[12] P. Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8:35-41, 1989.

[13] J. C. Laprie. Dependability evaluation of software systems in operation. *IEEE Trans. on Software Engineering*, 10(6):701-714, 1984.

[14] J. Ledoux. Availability modeling of modular software. *IEEE Trans. on Reliability*, 48(2):159-168, 1999.

[15] B. Littlewood. A reliability model for systems with Markov structure. *Applied Statistics*, 24(2):172-177, 1975.

[16] B. Littlewood. Software reliability model for modular program structure. *IEEE Trans. on Reliability*, 28(3):241-246, 1979.

[17] A. P. Mathur, P. Michielan, and M. Schiona. A comparison of techniques for component based reliability estimation of a software system. *Technical Report, SERC-TR-P*, 2001.

[18] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. on Software Engineering*, 18(1):33-43, 1992.

[19] E. Nelson. A statistical basis for software reliability. *TRW Software Series, TRW-SS-73-02*, 1973.

[20] A. Pasquini, A. N. Crespo, and P. Matrella. Sensitivity of reliability–growth models to operational profile errors *vs* testing accuracy. *IEEE Trans. on Reliability*, 45(4):531-540, 1996.

[21] C. V. Ramamoorthy and F. B. Bastani. Software reliability – status and perspectives. *IEEE Trans. on Software Engineering*, 8(4):354-371, 1982.

[22] M. Shooman. Structural models for software reliability prediction. In *Proc. 2nd Int'l Conference on Software Engineering*, pages 268-280, 1976.

[23] J. Voas, F. Charron, and K. Miller. Robust software interfaces: can COTS–based systems be trusted without them? In *Proc. 15th Int'l Conf. Computer Safety, Reliability, and Security*, pages 126-135, 1996.

[24] J. M. Voas. Certifying off–the–shelf software components. *IEEE Computer*, 31(6):53-59, 1998.

[25] C. Wohlin and P. Runeson. Certification of software components. *IEEE Trans. on Software Engineering*, 20(6):494-499, 1994.

[26] M. Xie and C. Wohlin. An additive reliability model for the analysis of modular software failure data. In *Proc. 6th Int'l Symp. Software Reliability Engineering*, pages 188-194, 1995.

[27] S. Yacoub, B. Cukic, and H. Ammar. Scenario–based reliability analysis of component–based software. In *Proc. 10th Int'l Symp. Software Reliability Engineering*, pages 22-31, 1999.