Architecture–based approach to reliability assessment of software systems

Katerina Goševa–Popstojanova *, Kishor S. Trivedi

Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708–0294, USA

Abstract

With the growing emphasis on reuse, software development process moves toward component–based software design. As a result, there is a need for modeling approaches that are capable of considering the architecture of the software and estimating the reliability by taking into account the interactions between the components, the utilization of the components, and the reliabilities of the components and of their interfaces with other components. This paper details the state of the architecture–based approach to reliability assessment of component based software and describes how it can be used to examine software behavior right from the design stage to implementation and final deployment. First, the common requirements of the architecture–based models are identified and the classification is proposed. Then, the key models in each class are described in detail and the relation among them is discussed. A critical analysis of underlying assumptions, limitations and applicability of these models is provided which should be helpful in determining the directions for future research.

Key words: Architecture–based software reliability, state–based models, path–based models, additive models

1 Introduction

A number of analytical models have been proposed to address the problem of quantifying the software reliability, one of the most important metrics of software quality. The variety of existing software reliability models can be

* Corresponding author. Tel.: +1-919-401-0299 ext 307; fax: +1-919-401-5589.
Email addresses: katerina@ee.duke.edu (Katerina Goševa–Popstojanova), kst@ee.duke.edu (Kishor S. Trivedi).

classified according to the several different classification systems [57]. The classification proposed in [45] is based primarily on the phase of software life cycle during which the models are applicable: debugging phase, validation phase, or operational phase.

A great deal of research effort in the past has been focused on modeling the reliability growth during the debugging phase. These so called black–box models treat the software as a monolithic whole, considering only its interactions with external environment, without an attempt to model internal structure. Usually, in these models no information other than the failure data is used. The common feature of black–box models is the stochastic modeling of the failure process, assuming some parametric model of cumulative number of failures over a finite time interval or of the time between failures. Failure data obtained while the application is tested are then used to estimate model parameters or to calibrate the model. For extensive survey on the black–box reliability growth models the reader is referred to books and papers such as [40,57,45,13,11].

With the growing emphasis on reuse, an increasing number of organizations are developing and using software not just as all–inclusive applications, as in the past, but also as component parts of larger applications. The existing black–box models are clearly inappropriate to model such a large component–based software system. Instead, there is a need for a modeling technique which can be used to analyze software components and how they fit together. The goal of the white–box approach is to estimate the 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 approach for quantitative assessment of software systems include the following:

- developing techniques to analyze the reliability and performance of applications built from reusable and COTS software components
- understanding how the system reliability/performance depend on its component reliabilities/performance and their interactions
- studying the sensitivity of the application reliability to reliabilities of components and interfaces
- guiding the process of identifying critical components and interfaces
- developing techniques for quantitative analysis that are applicable throughout the software life cycle.

Considering that software written from scratch is an exception, and that the modern software engineering practice is rather to make evolutions from existing software, a logical consequence is to enhance the predictions performed for a given product with field data relative to previous, similar, software products. When only the architecture, and not the code, is available, one can build a simulation model and conduct studies such as the ones designed to investigate

the impact of components on system reliability and performance for a given architecture. Managers can use this information to rethink the system's architecture and to plan resource allocation, for example. All such projections and decisions would be constantly updated as new testing data become available. Given the software architecture, information about the components and their interactions, one can perform calculations in an enlightening what–if analysis. What if a certain component were more or less reliable? How would that affect system reliability? Since architecture–based approach allows insight into the sensitivity of the total system to each component, it can be used to allocate effort to those components that are critical from the reliability or performance point of view. Architecture–based approach can be used to calculate system reliability from components information, or to stipulate a system reliability and calculate an allocated reliability of components. It can show the scale and the scope of effort needed to demonstrate required levels of components reliabilities. Finally, it can shed light on the reasonableness of building a system to desired levels of reliability on the basis of reusing a specific collection of components.

The aim of this paper is to provide an overview of the architecture–based approach to reliability assessment of software systems. The rest of the paper is organized as follows. The common requirements of the architecture–based models along with a classification are discussed in Section 2. The key models are classified and described in detail in Sections 3, 4, and 5. The relation among existing models is discussed in Section 6, while the models assumptions, limitations and applicability are discussed in Section 7. The concluding remarks are presented in Section 8.

2 Common requirements and classification

The main purpose of the following discussion is to focus attention on the framework within which the existing architecture–based software reliability models have been developed. Thus, different approaches for the architecture–based reliability estimation of the software are based on the following common steps.

Module identification
The basic entity in the architecture–based approach is the standard software engineering concept of a module. Although there is no universally accepted definition, a module is conceived as a logically independent component of the system which performs a well–defined function. This implies that a module can be designed, implemented, and tested independently. Module definition is a user level task that depends on the factors such as system being analyzed, possibility of getting the required data, etc. Module and component will be

used interchangeably in this paper.

Architecture of the software
The software behavior with respect to the manner in which the different modules of software interact is defined through the software architecture. Interaction occurs only by execution control transfer. In the case of sequential software, at each instant, control lies in one and only one of the modules. Software architecture may also include the information about the execution time of each module.

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 module or during the control transfer between two modules. The failure behavior of the modules and of the interfaces between the modules can be specified in terms of their reliabilities or failure rates (constant or time–dependent).

Combining the architecture with the failure behavior
Depending on the method used to combine the architecture of the software with the failure behavior, the literature contains three essentially different approaches: state–based approach, path–based approach, and additive approach. In the following sections the key models in each of the above classes are described in detail. The order of listed models is temporal based on when the paper was published.

3 State–based models

This class of models uses the control flow graph to represent the architecture of the system. It is assumed that the transfer of control between modules has a Markov property which means that given the knowledge of the module in control at any given time, the future behavior of the system is conditionally independent of the past behavior. The architecture of software has been modeled as 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 for which software runs that correspond to terminating execution can be clearly identified. The latter is well suited for continuously operating software applications, such that in real time control systems, where it is either difficult to determine what constitutes a run or there may be very large number of such runs if it is assumed that each cycle consists a run.

As suggested in [15], state–based models can be classified as either composite or hierarchical. The composite method combines the architecture of the soft-

ware 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.

Littlewood model [35]
This is one of the earliest, yet a fairly general architecture–based software reliability model.

Architecture. It is assumed that software architecture of continuously running application can be described by an irreducible semi Markov process, thus extending and generalizing the previous work [34] which describes software architecture with continuous time Markov chain. The program comprises a finite number of modules and the transfer of control between modules is described by the probability $p_{ij} = Pr\{\text{program transits from module } i \text{ to module } j\}$. The time spent in each module has a general distribution $F_{ij}(t)$ which depends upon i and j, with finite mean m_{ij}.

Failure behavior. When module i is executed, failures occur according to a Poisson process with parameter λ_i. The transfer of control between modules (interfaces) are themselves subject to failure; when module i calls module j there is a probability ν_{ij} of a failure occurring.

Solution method. The interest of the composite model is focused on the total number of failures of the integrated software system in time interval $(0, t]$, denoted by $N(t)$, which is the sum of the failures in different modules during their sojourn times, together with the interface failures. It is possible to obtain the complete description of this failure point process, but since the exact result is very complex, it is unlikely to be of practical use. The asymptotic Poisson process approximation for $N(t)$ is obtained under the assumption that failures are very infrequent. Thus, the times between failures will tend to be much larger than the times between exchanges of control, that is, many exchanges of control would take place between successive program failures. The failure rate of this Poisson process is given by

$$\lambda_S = \sum_i a_i \lambda_i + \sum_{i,j} b_{ij} \nu_{ij}$$

where

$$a_i = \frac{\pi_i \sum_j p_{ij} m_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}}$$

represents the proportion of time spent in module i, and

$$b_{ij} = \frac{\pi_i p_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}}$$

is the frequency of transfer of control between i and j. These terms depend only on parameters that characterize software architecture: transition probabilities p_{ij}, mean execution times m_{ij}, and steady–state probabilities of the embedded Markov chain π_i.

Cheung model [4]
This model considers the software reliability with respect to the module's utilization and their reliabilities.

Architecture. It is assumed that the program flow graph of a terminating application has a single entry and a single exit node, and that the transfer of control among modules can be described by an absorbing discrete time Markov chain with a transition probability matrix $P = [p_{ij}]$.

Failure behavior. Modules fail independently and the reliability of the module i is the probability R_i that the module performs its function correctly, i.e., the module produces the correct output and transfers control to the next module correctly.

Solution method. Two absorbing states C and F are added, representing the correct output and failure respectively, and the transition probability matrix P is modified appropriately to $\hat{P}$. The original transition probability p_{ij} between the modules i and j is modified into $R_i\, p_{ij}$, which represents the probability that the module i produces the correct result and the control is transferred to module 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 module i is considered by creating a directed edge to failure state F with transition probability $(1 - R_i)$. Thus, DTMC defined with transition probability matrix $\hat{P}$ is a composite model of the software system. 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 is not difficult to show that

$$S = I + Q + Q^2 + Q^3 + \ldots = \sum_{k=0}^{\infty} Q^k = (I - Q)^{-1},$$

so it follows that the overall system reliability can be computed as

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

Cheung's model is used in [55] to estimate the reliability of various heterogeneous software architectures such as batch–sequential/pipeline, call–and–return, parallel/pipe–filters, and fault tolerance. This model is also integrated in the Cleanroom Reliability Menager [44] whose aim is planning and certification of component–based software system reliability.

Laprie model [29]
This model is a special case of Littlewood model and the result, although obtained in a different way, agrees with those given in [34].

Architecture. The software system is made up of n components and the transfer of control between components is described by a continuous time Markov chain. The parameters are the mean execution time of a component i given by $1/\mu_i$ and the probability p_{ij} that component j is executed after component i given that no failure occurred during the execution of component i.

Failure behavior. Each component fails with constant failure rate λ_i.

Solution method. The model of the system is an $n+1$ state CTMC where the system is up in the states $i, 0 \leq i \leq n$ (componet i is executed without failure in state i) and the $(n+1)$th state (absorbing state) being the down state reached after a failure occurrence. The associated generator matrix between the up states $B = [b_{ij}]$ is given by

$$\begin{aligned} b_{ii} &= -(\mu_i + \lambda_i) \\ b_{ij} &= p_{ij}\mu_i, \qquad \text{for } i \neq j. \end{aligned}$$

The matrix B is seen as the sum of two generator matrices such that

- the execution process is governed by B' whose diagonal entries are equal to $-\mu_i$ and its off–diagonal entries to $p_{ij}\mu_i$
- the failure process is governed by B'' whose diagonal entries are equal to $-\lambda_i$ and off–diagonal entries are zero.

It is assumed that the failure rates are much smaller than the execution rates, and thus many exchanges of control are expected to take place before a failure occurs, that is, the execution process converges towards steady–state before a failure is likely to occur. Thus, the consequence of the assumption $\lambda_i \ll \mu_i$ is 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

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

where the steady–state probability vector $\pi = [\pi_i]$ is the solution of $\pi B' = 0$. This result has a simple physical interpretation keeping in mind that π_i is the proportion of time spent in state i when no failure occurs. The term $\pi_i \lambda_i$ can therefore be considered as the equivalent failure rate of component i.

Laprie and Kanoun in [31] considered the problem of modeling system's reliability and availability that covers both hardware and software. Considering the system made out of components they propose a variation of the above model such that the states of the CTMC are defined by more than one component under execution at a time. The system makes a transition between the states when there is a start or end of the execution of one or several components.

Kubat model [27]
This model considers the case of a terminating software application composed of n modules designed for K different tasks. Each task may require several modules and the same module can be used for different tasks.

Architecture. Transitions between modules follow a discrete time Markov chain such that with probability $q_i(k)$ task k will first call module i and with probability $p_{ij}(k)$ task k will call module j after executing in module i. The sojourn time during the visit in module i by task k has the pdf $g_i(k,t)$. Thus, the architecture model for each task becomes a semi Markov process.

Failure model. The failure intensity of module i is λ_i.

Solution method. The probability that no failure occurs during the execution of task k while in module i is

$$R_i(k) = \int_0^\infty \mathrm{e}^{-\lambda_i t}\, g_i(k,t)\ \mathrm{d}t.$$

The expected number of visits in module i by task k, denoted by $V_i(k)$, can be obtained by solving

$$V_i(k) = q_i(k) + \sum_{j=1}^{n} V_j(k)\, p_{ji}(k).$$

The probability that there will be no failure when running for task k can be

approximated by

$$R(k) = \prod_{i=1}^{n} [R_i(k)]^{V_i(k)}$$

and the system failure rate is calculated by

$$\lambda_s = \sum_{k=1}^{K} r_k [1 - R(k)]$$

where r_k is the arrival rate of task k.

Gokhale et al. model [17]
The novelty of this work lies in the attempt to determine the architecture of the software and the reliabilities of its components experimentally by testing the application using the regression test suite.

Architecture. The terminating application is described by an absorbing discrete time Markov chain. The trace data produced by the coverage analysis tool called ATAC [20] during the testing is used to determine the the transition probabilities p_{ij} between modules. The expected time spent in a module i per visit t_i is computed as a product of the expected execution time of each block and the number of blocks in the module.

Failure behavior. The failure behavior of each component is described by the enhanced non–homogeneous Poisson process model using a time–dependent failure intensity $\lambda_i(t)$ determined by block coverage measurements during the testing of the application together with the fault density approach.

Solution method. The expected number of visits to module i, denoted by V_i, is computed as in [27] by solving the following system of linear equations

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

where q denotes the initial state probability vector.

The reliability of module i, given time–dependent failure intensity $\lambda_i(t)$ and the total expected time $V_i t_i$ spent in the module per execution of the application, is given by

$$R_i = \mathrm{e}^{-\int_0^{V_i t_i} \lambda_i(t) \, \mathrm{d}t}.$$

Thus, the reliability of the overall application becomes

$$R = \prod_{i=1}^{n} R_i.$$

Ledoux model [32]
This recent work proposes an extension of the Littlewood model [34] to include the way failure processes affect the execution and to deal with the delays in recovering an operational state.

Architecture. A software composed of a set of components $\mathcal{C}$ is modeled by an irreducible continuous time Markov chain with transition rates q_{ij}.

Failure behavior. Two types of failures are considered: primary failure and secondary failure. The primary failure leads to an execution break; the execution is restarted after some delay. A secondary failure, as in [34,35,29], does not affect the software because the execution is assumed to be restarted instantaneously when the failure appears. Thus, for an active component c_i, a primary failure occurs with constant rate λ_i', while the secondary failures are described as Poisson process with rate λ_i''. When control is transferred between two components i and j then a primary (secondary) interface failure occurs with probability ν_{ij}' (ν_{ij}'').

Solution method. Following the occurrence of a primary failure a recovery state is occupied, and the delay of the execution break is a random variable with a phase type distribution. Denoting by $\mathcal{R}$ the set of recovery states, the state space becomes $\mathcal{C} \cup \mathcal{R}$. Thus, the CTMC that defines the architecture is replaced by a CTMC that models alternation of operational–recovery periods. The associated generator matrix defines the following transition rates: from c_i to c_j with no failure, from c_i to c_j with a secondary failure, from c_i to c_j with a primary failure, from recovery state i to recovery state j, and from recovery state i to c_j.

Based on the composite model described above the following measures are evaluated numerically: distribution function of the number of failures in a fixed mission, time to the first failure, point availability and failure intensity function.

Gokhale et al. reliability simulation approach [14]
This work demonstrates the flexibility offered by discrete event simulation to analyze component–based applications. The case study of a terminating application considers time–dependent failure rate of each component, finite debugging time, and fault–tolerant configurations for some of the components. Since the analytical model of such a system is intractable, discrete event simulation is an attractive alternative to study the influence of different factors on the

failure behavior of the application.

4 Path–based models

This class of models is based on the same common steps as the state–based models, except that the approach taken to combine the software architecture with the failure behavior can be described as a path–based since the system reliability is computed considering the possible execution paths of the program either experimentally by testing or algorithmically.

Shooman model [48]
This is one of the earliest models that considers reliability of modular programs, introducing the path–based approach by using the frequencies with which different paths are run.

Architecture. The model assumes the knowledge of the different paths and the frequencies f_i with which path i is run.

Failure behavior. The failure probability of the path i on each run, denoted by q_i, characterizes the failure behavior.

Method of analysis. The total number of failures n_f in N test runs is given by

$$n_f = \sum_{i=1}^{m} N f_i q_i$$

where Nf_i is the total number of traversals of path i. The system probability of failure on any test run is given by

$$q_0 = \lim_{N \to \infty} \frac{n_f}{N} = \sum_{i=1}^{m} f_i q_i.$$

Krishnamurthy and Mathur model [26]
This method for combining the architecture and failure behavior first involves computing the path reliability estimates based on the sequence of components executed for each test run, and then averaging them over all test runs to obtain an estimate of the system reliability.

Architecture. A sequence of components along different paths is observed using the component traces collected during the testing or simulation of the program behavior.

Failure behavior. Each component is characterized by its reliability R_m.

Method of analysis. A sequence of components along a path traversed for test case tc are considered as a series system. Assuming that individual components fail independently of each other it follows that the path reliability is the product of components reliabilities. The component trace of a program P for a given test case tc, denoted by $M(P, \text{tc})$, is the sequence of components m executed when P is executed against tc. Thus, the reliability of a path in P traversed when P is executed on test case tc $\in$ TS is given by

$$R_t = \prod_{\forall m \in M(P,\text{tc})} R_m.$$

The reliability estimate of a program with respect to a test set TS is

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

An interesting case occurs when most paths executed have components within loops and these loops are traversed a sufficiently large number of times. Then individual path reliabilities are likely to become low resulting in low system reliability estimates. By execution of a component an arbitrarily large number of times, it is easy to bring the system reliability estimate much below its true reliability if intra–component dependency is ignored. In this work intra–component dependency is modeled by "collapsing" multiple occurrences of a component on an execution path into k occurrences, where $k > 0$. k is referred as the degree of independence (DOI), such that complete dependence is modeled by setting DOI to 1 and complete independence by setting DOI to ∞. At this point it is not clear how one should determine a suitable value of DOI.

An alternative way to resolve the issue of intra–component dependency is proposed in [16]. The solution of dependency characterization of a component that is invoked inside a loop n times with a fixed execution time spent in the component per visit relies on the time–dependent failure intensity of a component. However, if the constant failure rate is assumed the solution reduces to the multiple independent executions of the component.

Yacoub, Cukic and Ammar model [59]
This reliability analysis technique is specific for component–based software whose analysis is strictly based on execution scenarios. A scenario is a set of component interactions triggered by specific input stimulus, and it is related to the concept of operations and run–types used in operational profiles [41].

Architecture. Using scenarios, a probabilistic model named Component Dependency Graph (CDG) is constructed. A node n_i of CDG models a component execution with an average execution time t_i. The transition probability p_{ij} is

associated with each directed edge e that models the transition from node n_i to n_j. CDG has two additional nodes, s–start node and t–termination node.

Failure behavior. The failure process considers component reliabilities R_i and transition reliabilities $(1-\nu_{ij})$ associated with a node n_i and with a transition from node n_i to n_j, respectively.

Method of analysis. Based on CDG a tree–traversal algorithm is presented to estimate the reliability of the application as a function of reliabilities of its components and interfaces. The algorithm expands all branches of the CDG starting from the start node. The breadth expansions of the tree represent logical "OR" paths and are hence translated as the summation of reliabilities weighted by the transition probability along 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 a terminating node (a natural end of an application execution) or when the summation of execution time of that thread sums to the average execution time of a scenario. The latter condition guaranties that the loops between two or more components don't lead to a deadlock.

5 Additive models

This class of models does not consider explicitly the architecture of the software. Rather, they are focused on estimating the overall application reliability using the component's failure data. It should be noted that these models consider software reliability growth. They are called additive models since under the assumption that component's reliability can be modeled by non–homogeneous Poisson process (NHPP) the system failure intensity can be expressed as the sum of component failure intensities.

Xie and Wohlin model [58]
This model considers a software system composed of n components which may be developed in parallel and tested independently. Assuming that a component failure is a system failure, from the reliability point view, the system can be considered as a series system. If the component reliabilities are modeled by NHPP with failure intensity $\lambda_i(t)$ then the system failure intensity is

$$\lambda_S(t) = \lambda_1(t) + \lambda_2(t) + \ldots + \lambda_n(t),$$

that is, the expected cumulative number of system failures by time t, known as mean value function, is given by

$$\mu_S(t) = \sum_{i=1}^{n} \mu_i(t) = \int_0^t \sum_{i=1}^{n} \lambda_i(\tau)\, \mathrm{d}\tau.$$

When this additive model is used the most immediate problem is that the starting time may not be the same for all components, that is, some components may be introduced into the system later in the testing process. In that case, the time of the corresponding mean value function and failure intensity have to be adjusted appropriately to consider different starting points for different components.

Everett model [10]
This approach considers the software made out of components, and addresses the problem of estimating individual component's reliability. Reliability of each component is analyzed using the Extended Execution Time (EET) model whose parameters can be determined directly from properties of the software and from the information on how test cases and operational usage stresses each component. Properties which affect component reliability are divided into static properties and dynamic properties. In addition, it must be characterized how usage stresses each component, both during testing periods and for periods when software is in operational use. For software the stressor of a component is the amount of processing time spent in a component. Therefore, this approach requires to keep track of the cumulative processing time per component.

The combined model for the overall system superimposes components reliabilities. When the underlying EET models for the components are non–homogeneous Poisson process models, the cumulative number of failures and failure intensity functions for the superposition of such models is just the sum of the corresponding functions for each component.

6 Models relation and unification

In published papers on architecture–based approach to software reliability a large number of variants have been proposed, mostly by ad hoc methods. These have frequently tended to obscure the unifying structural properties common to many such variants. The mathematical treatment of these models becomes evident once their common structure is exhibited. In this section we discuss the relation and uniformization of the existing architecture–based software reliability models.

6.1 State–based models

Architecture–based 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 system reliability is how to account for component's utilization, that is, for the stress imposed on each component during execution.

First, consider a terminating application whose architecture is described with DTMC. The absorbing DTMC can provide the average number of times each component is executed, denoted by V_i, as a measure of utilization of a component. Then $R_i^{V_i}$ can be considered as the equivalent reliability of component i that takes into account the component utilization. Thus, the system reliability becomes

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

which is the hierarchical approach to reliability estimation of the Cheung model [4].

Basically, this is the solution approach taken in [27]. The components reliabilities are estimated as the probability that no failure occurs during the execution time of component i

$$R_i = \int_0^{\infty} \mathrm{e}^{-\lambda_i t}\, g_i(t)\ \mathrm{d}t$$

and the embedded DTMC of the SMP that describes the software architecture provides the average number of times V_i each component is executed. Thus, the hierarchical treatment of this model reduces to the hierarchical treatment of the Cheung model.

The hierarchical approach in [17] estimates the component reliabilities considering time–dependent failure rates $\lambda_i(t)$ and the utilization of the components

through the cumulative expected time spent in the component per execution $V_i t_i$

$$R_i = \mathrm{e}^{-\int_0^{V_i t_i} \lambda_i(t)\ \mathrm{d}t}.$$

The system reliability then becomes $R = \prod_{i=1}^{n} R_i$.

As noted before, the common feature of the above models is that a software runs that correspond to terminating executions of software application can be clearly identified. Thus, the relevant measure is the reliability R of the single execution of software application, that is, so called probability of failure per demand $1 - R$.

Now, consider the continuously running applications. In [34] the moment generating function of number of failures $N(t)$ is derived from the composite model and it is concluded that the analytically tractable results can not be obtained. However, the asymptotic analysis of the model led to Poisson process with parameter

$$\lambda_S = \sum_i \pi_i \left[\lambda_i + \sum_{j \neq i} q_{ij} \nu_{ij} \right]$$

where $\pi = [\pi_i]$ is the equilibrium vector of the irreducible CTMC with transition rate matrix $Q = [q_{ij}]$, i.e. $\pi Q = 0$. The term in the parentheses $\Lambda_i = \lambda_i + \sum_{j \neq i} q_{ij} \nu_{ij}$ can be interpreted as a failure rate of component i that includes the internal component failure rate λ_i and the failure rate due to the failures that occur during interaction with other components. Thus, $q_{ij} \nu_{ij}$ gives the interface failure rates characterizing the failures occurring during interaction of components i and j. Summing over j, therefore, gives the the total interface failure rate from state i. Since, the steady–state probability π_i represents the average proportion of time spent in state i in the absence of failure, the term $\pi_i \Lambda_i$ can therefore be considered as the equivalent failure rate of component i that takes into account the component utilization.

The model presented in [29] is the special case of [34] which considers only component failures. In this case the equivalent failure rate

$$\lambda_S = \sum_i \pi_i \lambda_i$$

is obtained using the hierarchical method based on probabilistic arguments. It is obvious that this is a special case of λ_S obtained in [34] for $\nu_{ij} = 0$.

These results deserve 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 additive models.

Further, consider the probability that there will be no failure up to time t, that is, the system reliability

$$R(t) = \mathrm{e}^{-\lambda_S\, t} = \mathrm{e}^{-\sum_{i=1}^{n} \pi_i \Lambda_i\, t} = \prod_{i=1}^{n} \mathrm{e}^{-\Lambda_i\, \pi_i t} \ .$$

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 [31]. From the reliability point of view, this leads to the usual equation for a series system with a number of subsystems with exponentially distributed time to failure.

Basically, the above models are the special cases of the versatile Markov point process introduced by Neuts in [43] which was recently shown to be equivalent to Batch Markovian Arrival Process [37]. Since this is a rich class of point processes for which many relevant probability distributions, moment and correlation formulas can be obtained in forms which are computationally tractable, they have been used extensively to model arrival processes in queuing theory.

The construction of the versatile Markov point process is as follows. Consider an $n+1$ state CTMC with n transient states and one absorbing state. The infinitesimal generator Q^* obtained after deletion of the absorbing state is irreducible and describes the CTMC obtained by resetting the original CTMC instantaneously using the same initial probabilities, whenever an absorption into state $n+1$ occurs. The times of absorption (and resetting) form a renewal process with the underlying phase–type distribution.

The point process is constructed by starting with this phase–type renewal process as a substratum. Three types of arrival epochs which are related to the evolution of the phase–type renewal process are considered. There are Poisson arrivals with arbitrary batch size distributions during sojourns in the states of the CTMC governing the renewal process. The arrival rates of the Poisson process and the batch size distributions may depend on the state of the CTMC. The underlying CTMC can change states either with or without a corresponding renewal. Each time the process changes states there is a batch arrival (the batch size may be 0) where the batch size can depend on the states before and after the change as well as whether or not a renewal occurred.

Let $N(t)$ and $X(t)$ denote respectively the number of arrivals in $(0,t]$ and the state of the irreducible CTMC defined by Q^* at time t. The process $\{N(t), X(t), t \geq 0\}$ is a CTMC with state space $\{k \geq 0\} \times \{1, \ldots n\}$. The close relation of the BMAP with the finite CTMC results in the matrix–analytic approach that substantially reduces the computational complexity of the algorithmic solution.

It is obvious that [34,29,32] are the special cases of the BMAP. Thus, in [34,32] only the arrivals with batch size 1 are considered which leads to the special case of BMAP, so called Markovian Arrival Process (MAP) [37]. The model presented in [29] that considers only the Poisson arrivals during sojourn times in the states of the Markov process is doubly stochastic Poisson process known as Markov modulated Poisson process (MMPP) [37].

Ledoux model [32], which generalizes [34,29], is a composite model. The solution method used in this paper is based on the the matrix–analytical approach that was pioneered by Neuts [43]. Thus, many relevant quantities connected with the failure point process are derived in forms which are computationally tractable. These include the distribution function of the number of failures $N(t)$ in $(0,t]$, waiting time the to first failure, i.e reliability $R(t) = Pr\{T > t\} = Pr\{N(t) = 0\}$, expected number of failures $E[N(t)]$ in $(0,t]$, and failure intensity function $h(t)$.

It is well known that interarrival times of BMAP are phase–type, and that a phase–type distribution with an irreducible matrix is asymptotically exponential with the eigenvalue of maximal real part λ_S of matrix as parameter. The scalar λ_S, known as fundamental arrival rate for the point process [37], can be interpreted as the asymptotic failure rate of the system when $t \to \infty$. In other words, when $t \to \infty$ the failure intensity function $h(t)$ becomes a constant value $h(\infty) = \lambda_S$. Using this approach in [32] the Poisson approximation is compared with the transient solution for the point process of the composite model. If only secondary failures are considered the asymptotic failure rate reduces to the one obtained in [34], i.e. [29] which is the parameter of the Poisson distribution that approximates the distribution for the number of failures $N(t)$ in $(0,t]$.

Let us now consider Littlewood model [35] which assumes that software architecture can be described with SMP. The exact non–asymptotic analysis of the composite model becomes very difficult. However, under the assumption that many exchanges of control will take place between successive program failures the failure point process is asymptotically the Poisson process, as stated in [35]. The equivalent failure rate of the asymptotic Poisson process can be derived using the hierarchical method based on probabilistic arguments as follows. The solution of the architecture model can be obtained using the well known limiting behavior of semi Markov processes [28]. Thus, architecture

model provides the limiting proportion of time spent in each module i and limiting number of i to j transfers of control per unit of time, denoted by a_i and b_{ij} respectively, as the measures of components and interfaces utilization. Then, the failure behavior is superimposed onto the solution from architecture model to obtain the equivalent failure rate

$$\lambda_S = \sum_i a_i \lambda_i + \sum_{i,j} b_{ij} \nu_{ij}.$$

6.2 Path–based models

Similar to state–based models, path–based models consider the software architecture explicitly and assume that components fail independently. However, the method of combining software architecture with components and interfaces failure behavior is not analytical. First, the sequence of components executed along each path is obtained either experimentally by testing [26] or algorithmically [59] and the path reliability is obtained by multiplying the reliabilities of the components along that path. Then, the system reliability is estimated by averaging path reliabilities over all paths. Thus, these models account for each component utilization along each path, as well as among different paths.

The difference between the state–based approach and path–based approach 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 [26] or the depth traversal of each path is terminated using the average execution time of the application [59].

6.3 Additive models

Additive models [58,10] consider software testing phase and assume that each component reliability can be modeled by non–homogeneous Poisson process. They can be seen as a superposition of non–homogeneous Poisson processes. It is well known that if the component failure processes are modeled by a 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. However, the models differ in the following.

In [58] the time of the corresponding functions is adjusted appropriately to consider the time when different components are incorporated into the system. This model does not consider the software architecture, that is, different

utilization of the components.

The model used for capturing component's reliability growth in [10] includes the information about the relative usage stress imposed on each component as a parameter of the model. Also, the cumulative execution time spent in each component obtained during the testing is used to compute the cumulative number of failures and failure intensity function for the individual components. Therefore, this model considers the utilization of software components, that is, the software architecture implicitly. In other words, this approach is closely related to the approach for estimating components reliabilities presented in [17] that explicitly takes into account the software architecture.

7 Models assumptions, limitations and applicability

The benefit of the architecture–based approach is evident in the context of software system that is developed as a heterogeneous mixture of newly developed, reused and COTS components. However, this approach appears to add complexity to the models and to data collection as well. Many questions related to the architecture–based approach to quantitative assessment of software systems are still unanswered, and more research in this area is needed, in particular when it comes to the issues indicated below.

Level of decomposition

There is a trade off in defining the components. Too many small components could lead to a large state space which may pose difficulties in measurements, parametrization, and 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.

Numerous papers deal with the architecture–based reliability modeling, but just a few experimental studies has been published so far. The choices made in these experiments for the level of decomposition are very diverse. For example, two decompositions of the telephone switching software system were considered in [22]: decomposition into four groups according to the main functions of the system, and decomposition into four classes according to the failure consequences. The experiment reported in [26] uses the decomposition based on the standard components of the Unix utility *grep*. In [17] the decomposition of SHARPE, a tool used to solve stochastic models of reliability, performance and performability [46], is based on the file level; each of the 30 files is regarded as a single component. The case study presented in [59] uses the application

that simulates the behavior of waiting queues which is composed of six reused components.

Estimation of individual component reliabilities

Most of the papers on architecture–based reliability estimation assume that component reliabilities are available, that is, ignore the issue of how they can be determined. Assessing the reliability of individual 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.

Of course, one might argue that the reliability growth models can be applied to each software component exploiting component's failure data. For example, Kanoun et al. [22] applied the model based on nonhomogeneous Poisson process, the hyperexponential model, in order to estimate the stationary failure rates of components. This study has been conducted on the information concerning the failures of a telephone switching system which were observed over three years including the validation and operational phases. Gokhale et al. [17] used the enhanced NHPP model, proposing a method for determining component's time–dependent failure intensity based on block coverage measurement during the testing. Everett [10] identified guidelines for estimating reliability of the newly developed components based on the Extended Execution Time model whose parameters are related to the component's static and dynamic properties and the usage of each component.

However, it is not always possible to use software reliability growth models for estimating the individual component's reliability. A difficulty could arise due to the scarcity of failure data. Predictions based on failure data are of interest only if enough data are available in order to be statistically representative. Further difficulties are due to the fact that the underling assumptions of software reliability growth models such as random testing performed accordingly to the operational profile and the independence of successive testing runs can be seriously violated during the unit testing phase [18].

Another class of models estimate reliability from explicit consideration of non–failed executions, possibly together with failures [42,38,44,36]. Such models can be used to make reliability estimations of the software based on the results of testing performed in its validation phase. In this phase no changes are made to the software, and these models can be termed as stable reliability models. In this context, testing is not a development activity for discovering faults, but an independent assessment of the software execution in representative operational environment. The problem which arises with these models is the number of executions necessary for estimating reliability levels which are commensurate with reasonable expectations. The probabilistic assessment

of software reliability to levels required in the safety–critical applications, e.g. failure rate 10^{-9}/h or 10^{-5} failure probability per demand, is currently out of reach [3], since random testing would require decades of testing before it could establish a reasonable statistical confidence in the reliability estimate. It is generally agreed that a practical current limit in the assessment of a failure rate before operational use lies in the range 10^{-2}/h - 10^{-4}/h [30].

Several other techniques for estimating component's reliability have been proposed. Krishnamurthy et al. [26] used the method of seeding faults. Voas [53] examined a method to determine the quality of COTS components using black–box component testing and system–level fault injection. However, a difficulty with fault–based techniques is that they are only as powerful as the range of fault classes that they simulate [52].

Estimation of interface reliabilities

There seems to be little information available about interface failures, apart from general agreement that they exist separately from component failures which are revealed during the unit testing. The interface between two components could be another component, a collection of global variables, a collection of parameters, a set of files, or any combination of these. In practice, one needs to estimate the reliability of each interface. When an interface is a program, any of the methods mentioned above can be used to estimate their reliabilities. However, when an interface consists of items such as global variables, parameters, and files, it is not clear how to estimate reliability. Some explanation and analysis about the interfaces between components has been performed by Voas et al. [51]. Also, method for integration testing proposed by Delamaro et al. [7] seems promising for estimating interface reliabilities.

Validity of Markov assumption

All state–space models assume that the next component to be executed will depend probabilistically on the present component only and is independent of the past history, that is, they assume that the embedded Markov chain is a first–order chain. However, this needs to be justified which is common to all situations where a Markov assumption is made. For example, the hypothesis that the chain is of a given order is tested in [19] for the stochastic model that characterizes the interactive workload.

If the probability to find the system in given state (executing a specified component) does not depend only on the last executed component but on a given number of components which were executed before the specified state was entered then the system can be described by a model of higher order. It is well known that a higher order Markov chain can be represented as a first order chain by redefining the state space appropriately [6]. Consider a

second order Markov chain and let the pair of successive states i and j define a composite state (i, j). Then the transition probability to composite state (j, k) given the composite state (i, j) is p_{ijk}. The composite states are easily seen to form a chain with n^2 states with certain transition probabilities 0. This representation is useful because the results for the first order Markov chain can be carried over. Higher order Markov chain can be handled in a similar way. If the system can be described by a model of order r the state space is defined by a set of all possible r-tuples and the number of elements in the state space becomes n^r.

The higher order Markov chain enables us to consider the situations when the execution time of a component and its failure behavior are dependent on the number of components along the path from which the control was transferred. This fact was recognized in [32]. However, the size of the state space grows fast with the number of components which makes model specification and analysis difficult and error–prone process, requires considerable amount of data, and imposes difficulties in both measurements and parametrization.

Estimation of transition probabilities

In architecture–based approach, one must model the interaction of all components. In well designed system, interaction among components is limited. In most cases many interactions among components will be impossible, that is, transition probability matrix will be sparse. During the early phases of software life cycle, each component could be examined to find with which components it cannot interact. The remaining, non–zero transition probabilities may be available by analyzing program structure and using known operational profiles. More likely it will be necessary to estimate the transition probabilities. During the design phase, before actual development and integration phases, it can be done by simulation. During the integration phase, as new data become available, the estimates have to be updated thereby improving the predictions. Fortunately, good estimates can be obtained very early due to the high speed of control exchanges. The estimation of transition probabilities must be as accurate as possible since the error in this process will definitely affect system reliability and components sensitivities.

The following two papers demonstrate two different approaches for estimating transition probabilities. In [59] first the estimates of the probabilities of execution of different scenarios are obtained based on the operational profile. Then, using the analysis scenarios the transitions probabilities are calculated. In [17] the transition probabilities are computed based on the execution counts extracted from the ATAC trace files obtained during coverage testing.

Operational profile

Software reliability models assume that random test selection is guided accordingly to the user's operational profile [41]. Test selection aimed at finding faults, increasing various structure coverages or demonstrating different functional requirements are not representative of the user's operational profile and might lead to reliability estimate different from one perceived by the user of the system. Further, 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. The most radical change is to remove or add states and associated transitions, which corresponds to a major architectural change. A less radical change is to remove or add transitions or change the transition probabilities without changing the components themselves.

Also, care must be taken to ensure that the change of operational profile is considered in assessing component's reliability. In [56] it is suggested that a reliability measure of a component developed for reuse is attached to the actual operational profile used during certification. Then, it is necessary to compare the certified operational profile with the environment in which the component is to be reused. If the component has not been certified for the operational profile of the new system, a new certification must be performed.

Considering failure dependencies among components and interfaces

Without exception the existing models assume that the failure processes associated across different components are mutually independent. When considered, the interface failures are also assumed to be mutually independent and independent of the component's failure processes. However, if the failure behavior of a component is affected in any way by the previous component being executed, or by the interface between them, these assumptions are no longer acceptable. This dependence can be referred as inter–component dependence [26]. One way to encounter for inter–component dependence in the case of state space models would be to consider Markov chain of higher order as discussed above. Intra–component dependence can arise, for example, when a component is invoked more than once in a loop by another component. An attempt to address intra–component dependence was made in [26,16].

Extracting software architecture

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 static architectural information can be either parser–based or lexically–based [39], while the system's dynamic behavior can be captured using profilers or test coverage tools. Recently, a workbench for architectural extraction, called *Dali*, that fuses different architectural views was developed [23].

Next a brief description of the tools that are currently used at Duke University to extract the dynamic view of the software architecture is presented.

- The GNU profiler *gprof* [60] for C and C++ programs can be used to obtain a flat profile and a call graph for the program. The flat profile shows the time spent in each function and the number of times the function was visited. The call graph shows for each function which functions called it, which functions it calls, how many times, and an estimate of the time spent in the subroutines of each function. From the call graph, the architecture of the application can be specified in terms of a control flow graph. Information on the number of times each function calls other functions can be used to obtain the transition probabilities from one function to another. Some architecture–based models also need information on the time spent in each module which can be obtained from the flat profile.
- χATAC [20] is a coverage testing tool that is part of the Software Visualization and Analysis Toolsuite (χSuds) [61] developed by Telcordia Technologies. It can report various code coverage measures that help evaluate the quality of a test suite. It also provides a command interface that can be used to query the log files produced in the process of obtaining code coverage information, thus providing information on visit counts at a level as low as a block of code. Information on the number of times each block calls other blocks can also be obtained. From the knowledge of visit counts and sequence of transitions, the architecture of the software can be extracted at different levels of granularity. χATAC is especially useful when the coverage information from testing is used to estimate component reliabilities, as both coverage and architecture information can be obtained using the same tool [17].
- The ATOM toolkit is part of the Compaq Tru64 Unix (formerly Digital Unix) [62] operating system. It consists of a programmable instrumentation tool and several packaged tools. Prepackaged tools that are available and useful for extracting the architecture of an application include: obtaining a flat profile of an application that shows the execution time for a procedure; counting the number of times each basic block is executed, number of times each procedure is called, and the number of instructions executed by each procedure; printing the name of each procedure as it is called. These tools can be used to specify the architecture of an application by means of a call graph, the transition probabilities from one component of the software to another, or time spent in each component.

  ATOM has the following favorable features. It does not need the source code of the application, but rather operates on object modules. Therefore, the use of ATOM is independent of any compiler and language. Also, the instrumentation and analysis routines that form the ATOM toolkit can be user–defined which can be useful in obtaining data required to parameterize a wider range of architecture–based models. However, ATOM is only available as part of the Compaq Tru64 UNIX operating system.

Sensitivity analysis

Since the architecture–based approach allows insight into the contribution made by each component to the unreliability of the whole system, it can be used to perform sensitivity analysis. Although this advantage is recognized by many researchers [26,17,59], the method for computing the sensitivity of system reliability with respect to a given component reliability is developed only for the model presented in [4]. This approach to sensitivity analysis is used in [44] for reliability allocation to each component which is based on the target reliability for the entire system and the sensitivity of the system to the component.

The sensitivity analysis can help to identify the critical components which have the greatest impact on system reliability and performance. This information can be used for planning and certification activities during different phases of software life cycle.

Considering multiple software quality attributes

Architecture–based approach for quantitative assessment of software systems is mainly focused on reliability evaluation. However, it has a potential for performance evaluation since most of the existing architecture–based models consider component execution times. For example, the average execution time of each component is considered in [17,48,59], the exponentially distributed execution time is considered in [34,29,32], and the execution time with general distribution in [35,27]. Although these models recognize the importance of component utilization from the reliability point of view, none of them considers the formal analysis of software performance.

Performance as a software quality attribute refers to the timeliness aspects of how software system behaves, that is, characterizes the timeliness of the service delivered. First, consider a terminating application. A possible measure of software performance is the expected execution time of the application which can be computed as [50]

$$\sum_i V_i t_i$$

where the expected number of visits to state i is already computed for the purpose of reliability estimation while the expected time spent in a component per visit t_i is obtained experimentally in [17] or can be computed as $t_i = \int_0^\infty t\, g_i(t)\, dt$ in [27].

Next, consider a continuously running application whose architecture is described with SMP as in Littlewood model [35]. If the SMP starts in state j, then one cycle is completed every time the SMP returns to j. The measure of

performance could be the expected time of one cycle given by [28]

$$\frac{1}{\pi_j}\sum_i \pi_i m_i = \frac{1}{\pi_j}\sum_i \pi_i \sum_j p_{ij} m_{ij}$$

where $m_i = \sum_j p_{ij} m_{ij}$ is the mean time spent in state i during each visit.

The relative importance of performance and reliability requirements will differ depending on the system requirements and typical usage. Sometimes performance and reliability issues can be addressed separately, but sometimes their interactions and corresponding tradeoffs need to be considered in order to arrive at a better overall system.

Depending on the quaility attributes of interest different qualitative and quantitative techniques can be used to conduct the analysis. These techniques have evolved in separate communities, each of its own vernacular and point of view. Thus, this paper presents an overview from the perspective of Software reliability engineering community, while Smith and Williams in [49] describe the analysis framework called Software performance engineering that provides techniques to the management of performance concerns in the development of software based systems. These communities are focused on different quality attributes in isolation, without considering the impact on other attributes and tradeoffs between multiple conflicting attributes.

The analysis of multiple software quality attributes and their tradeoffs is an important area where much work needs to be done. The first step in that direction was made by Barbacci et al. [1]. They introduce a generic taxonomy of different attributes, including an attempt of developing unifying approach for reasoning about multiple software quality attributes. This work led to the Architecture Tradeoff Analysis Method (ATAM), developed at the Software Engineering Institute [2,24], which addresses the problem of architecture analysis based on multiple quality attributes such as performance, availability and security.

Considering different architectural styles

Today's software applications are far more complex, frequently run on two or more processors, under different operating systems, and on geographically distributed machines. As a result, there is a proliferation of research activities focused on software architecture [1]. Ground–work for studying software architecture was laid by Garlan and Show [47]. They identified a useful set of software architectural styles. An architectural style is determined by a set

[1] See, for example, the April 1995 issue of *IEEE Transaction on Software Engineering*, November 1995 and January/February 1997 issues of *IEEE Software*.

of component types, a topological layout of these components indicating their interrelationships, and a set of interaction mechanisms that determine how they coordinate. For example, some systems have a pipe and filter style (e.g. Unix like) while others are object oriented, data centered (e.g. transactional database systems) or event systems (e.g. objects broadcast events through a request broker). These styles differ both in the kinds of components they use and in the way those components interact with each other. Components define the application level computations and data stores of a system. Examples include clients, servers, filters, databases, and objects. Interaction between those components can be as simple as procedure calls, pipes, and event broadcast, or much more complex, including client–server protocols, database accessing protocols, etc.

Because architectural decisions are usually made early in the life–cycle, they are the hardest to change and hence the most critical and far–reaching. Therefore software architecture plays a major role in determining system quality and maintainability. If software engineering is to mature as an engineering discipline, standardized architectural styles will have to be developed, along with the methods for their quantitative assessment. This will help the designers to select the architectural style that is most appropriate for the system in hand.

Modeling and evaluation that encompasses both hardware and software

The users of computing systems are interested in obtaining results from modeling and evaluation of systems, composed of both hardware and software. Even though reliability evaluation of combined hardware and software systems is not yet common practice, some relevant papers have been published. Thus, the work presented in [31] is aimed at extending the classical reliability theory in order to be interpreted from both hardware and software viewpoints. Further, modeling and evaluation of the fault–tolerant systems that account for the hardware and software components' behavior and consider their interactions explicitly have been presented in [8,9,12,21].

The models of nonredundant sequential software systems considered throughout this paper only consider the software part of the system assuming that it is possible to separate the software system from its hardware platform. This distinction helps to separately define software and hardware models and then to solve their combination. It also improves the portability of the models. For software applications that run in a distributed environment emphasis should be placed on clear definition of the interactions between the software and hardware components. Additional challenges posed by distributed systems include concurrent execution of components, non–instantaneous transfer of control, failures associated with deadline violations due to the communication overheads, communication errors, node failures, etc.

Deriving the model from specification

The task of reliability and performance modeling requires insight and skill. However, implementing the mappings from specification languages into appropriate reliability and performance modeling formalisms should minimize the effort and the cost. One of the first approaches in this direction is the work presented in [54] which proposes a procedure of transforming a specification written in the formal specification language Estelle into a stochastic model which then can be used to carry out the reliability and performance analysis.

More recently there has been growing interest to close the gap between commercial design tools and quantitative evaluation of software systems. These efforts are mainly focussed on the Unified Modeling Language (UML) that has emerged as a widely accepted standard notation for software design in the last two years. UML provides system architects working on object oriented analysis and design with one consistent language for specifying, visualizing, constructing, and documenting the artifacts of software systems. A number of recent papers consider the transformation of the UML specification (enhanced by timing constraints) into the stochastic performance models [25,33,5]. We believe that the further advances of the methodological approaches for systematic and automatic generation of reliability and performance models from software specifications will bring reliability and performance evaluation into the software design process.

8 Conclusion

In this paper, the overview of the architecture–based approach to quantitative assessment of component–based software systems is presented. Based on the methods used to describe the architecture of the software and to combine it with the failure behavior of the components and interfaces, three classes of models are identified. The state–based models describe the software architecture as a discrete time Markov chain, continuous time Markov chain, or semi Markov process, and estimate the software reliability analytically by combining the architecture with failure behavior. The models from the class called path–based compute the software reliability considering the possible execution paths of the program either experimentally by testing or algorithmically. The third modeling approach, called additive, does not consider the architecture explicitly. Rather, it is focused on estimating the time–dependent failure intensity of the application using the component's failure data.

Further, the paper presents an attempt to relate and unify the existing architecture–based models. Finally, the underlying assumptions of the architecture–based

models are discussed in order to provide an insight into the usefulness and limitations of such models that would be helpful in determining the directions for future research.

Our goal in this paper was to present the state of the research and practice in the architecture–based approach to quantitative assessment of software reliability and to point out that the knowledge that already exists in the various research communities need to be drawn on to enhance current software engineering practices.

Acknowledgements

We acknowledge the financial support of NSF under the grant EEC-9714965 (IUCRC TIE grant between Duke CACC and Purdue SERC). We also acknowledge the financial support of the Department of Defense, Alcatel, Stratus, and Telcordia by means of a CACC Core project. Thanks are also due to Srinivasan Ramani for his help with this paper.

References

[1] M.Barbacci, M.Klein. T.Longstaff, C.Weinstock, Quality attributes, CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

[2] M.Barbacci, J.Carriere, P.Feiler, R.Kazman, M.Klein, H.Lipson, T.Lingstaff, C.Weinstock, Steps in an architecture tradeoff analysis method: quality attribute models and analysis, CMU/SEI-97-TR-29, Software Engineering Institute, Carnegie Mellon University, 1997.

[3] R.W.Butler, G.B.Finelli, The infeasibility of quantifying the reliability of life–critical real–time software, *IEEE Trans. on Software Engineering*, 19 (1) (1993) 3-12.

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

[5] V.Cortellessa, R.Mirandola, Deriving a queueing network based performance model from UML diagrams, in: *Proc. 2nd Int'l Workshop on Software and Performance (WOSP2000)*, 2000, pp. 58-70.

[6] D.R.Cox, H.D.Miller, *The Theory of Stochastic Processes*, Chapman and Hall, 1990.

[7] M.Delamaro, J.Maldonado, A.P.Mathur, Integration testing using interface mutations, in: *Proc. 7th Int'l Symp. on Software Reliability Engineering (ISSRE'96)*, 1996, pp. 112-121.

[8] J.B.Dugan, A.A.Doyle, F.A.Patterson–Hine, Simple models of hardware and software fault tolerance, in: *Proc. Annual Reliability and Maintainability Symp.*, 1994, pp. 124-129.

[9] J.B.Dugan, M.Lyu, System reliability analysis of N version programming application, *IEEE Trans. on Reliability*, 43 (4) (1994) 513-519.

[10] W.Everett, Software component reliability analysis, in: *Proc. Symp. Application–Specific Systems and Software Engineering Technology (ASSET'99)*, 1999, pp. 204-211.

[11] W.Farr, Software reliability modeling survey, in: M.R.Lyu, (Ed.), *Handbook of Software Reliability Engineering*, McGraw–Hill, 1996, pp. 71-117.

[12] S.Garg, Y.Huang, C.M.R.Kintala, K.S.Trivedi, S.Yajnik, Performance and reliability evaluation of passive replication schemes in application level fault tolerance, in: *Proc. 29th Int'l Symp. on Fault–Tolerant Computing (FTCS 29)*, 1999, pp. 322-329.

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

[14] S.Gokhale, M.Lyu, K.Trivedi, Reliability simulation of component based software systems, in: *Proc. 9th Int'l Symp. Software Reliability Engineering (ISSRE'98)*, 1998, pp. 192-201.

[15] S.Gokhale, K.Trivedi, Structure–based software reliability prediction, in: *Proc. 5th Int'l Conf. Advanced Computing (ADCOMP'97)*, 1997, pp. 447-452.

[16] S.Gokhale, K.Trivedi, Dependency characterization in path–based approaches to architecture based software reliability prediction, in: *Proc. Symp. Application–Specific Systems and Software Engineering Technology (ASSET'98)*, 1998, pp. 86-89.

[17] S.Gokhale, W.E.Wong, K.Trivedi, J.R.Horgan, An analytical approach to architecture based software reliability prediction, in: *Proc. 3rd Int'l Computer Performance & Dependability Symp. (IPDS'98)*, 1998, pp. 13-22.

[18] K.Goševa–Popstojanova, K.Trivedi, Failure correlation in software reliability models, *IEEE Trans. on Reliability*, 49 (1) (2000) 37-48.

[19] G.Haring, On stochastic models of interactive workloads, in: *Performance'83*, 1983, pp.133-152.

[20] J.R.Horgan and S.London, ATAC: A data flow coverage testing tool for C, in: *Proc. 2nd Symp. Assessment of Quality Software Development Tools*, 1992, pp. 2-10.

[21] K.Kanoun, M.Borrel, T.Morteville, A.Peytavin, Availability of CAUTRA, a subset of the French air traffic control system, *IEEE Trans. on Computers*, 48 (5) (1999) 528-535.

[22] K.Kanoun, T.Sabourin, Software dependability of a telephone switching system, in: *Proc. 17th Int'l Symp. on Fault–Tolerant Computing (FTCS 17)*, 1987, pp. 236-241.

[23] R.Kazman, J.Carriere, Playing detective: Reconstructing software architecture from available evidence, *Journal of Automated Software Engineering*, 6 (2) (1999) 107-138.

[24] R.Kazman, M.Klein, M.Barbacci, T.Lingstaff, H.Lipson, J.Carriere, The architecture tradeoff analysis method, in: *Proc. of IEEE Intr. Conf. on Engineering of Complex Computer Systems (ICECCS'98)*, 1998, pp. 68-78.

[25] P.King, R.Pooley, Derivation of Petri net performance models from UML specifications of communications software, in: *Proc. 11th Int'l Conf. on Tools and Techniques for Computer Performance Evaluation (TOOLS 2000)*, Lecture Notes in Computer Science, vol.1786, Springer, Berlin, 2000, pp. 262-276.

[26] S.Krishnamurthy, 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 (ISSRE'97)*, 1997, pp. 146-155.

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

[28] V.G.Kulkarni, *Modeling and Analysis of Stochastic Systems*, Chapman & Hall, 1995.

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

[30] J.C.Laprie, Dependability of computer systems: concepts, limits, improvements, in: *Proc. 6th Int'l Symp. Software Reliability Engineering (ISSRE'95)*, 1995, pp. 2-11.

[31] J.C.Laprie, K.Kanoun, X–ware reliability and availability modeling, *IEEE Trans. on Software Engineering*, 18 (2) (1992) 130-147.

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

[33] C.Lindemann, A. Thümmler, A.Klemm, M.Lohmann, O.P.Waldhordt, Quantitative system evaluation with DSPNexpress 2000, in: *Proc. 2nd Int'l Workshop on Software and Performance (WOSP2000)*, 2000, pp. 12-17.

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

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

[36] B.Littlewood, D.Wright, Some conservative stopping rules for operational testing of safety–critical software, *IEEE Trans. on Software Engineering*, 23 (11) (1997) 673-683.

[37] D.M.Lucantoni, New results on the single server queue with a batch Markovian arrival process, *Stochastic Models*, 7 (1), (1991) 1-46.

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

[39] G.Murphy, D.Notkin, E.Lan, An empirical study of static call graph extractors, in: *Proc. 18th Int'l Conference on Software Engineering (ICSE 18)*, 1996, pp. 90-99.

[40] J.D.Musa, A.Iannino, K.Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGrow–Hill, 1987.

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

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

[43] M.F.Neuts, A versatile Markovian point process, *Journal of Applied Probability*, 16 (1979), 764 - 779.

[44] J.H.Poore, H.D.Mills, D.Mutchler, Planning and certifying software system reliability, *IEEE Software*, 10 (1) (1993) 88-99.

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

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

[47] M.Shaw, D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice–Hall Inc., 1996.

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

[49] C.Smith, L.Williams, Software performance engineering: A case study including performance comparison with design alternatives, *IEEE Trans. on Software Engineering*, 19 (7) (1993) 720-741.

[50] K.S.Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Prentice–Hall, 1982.

[51] J.Voas, F.Charron, K.Miller, Robust software interfaces: Can COTS–based systems be trusted without them?, in: *Proc. 15th Int'l Conf. Computer Safety, Reliability, and Security (SAFECOMP'96)*, 1996, pp. 126-135.

[52] J.M.Voas, C.C.Michael, K.W.Miller, Confidently assessing a zero probability of software failure, *High Integrity Systems*, 1 (3) (1995) 269-275.

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

[54] C.Wang, K.S.Trivedi, Integration of specification form modeling and specification for system design, in: *Proc. 14th Int'l Conf. Application and Theory of Petri Nets*, Lecture Notes in Computer Science, vol.691, Springer, Berlin, 1993, pp. 473-492.

[55] W.Wang, Y.Wu, M.Chen, An architecture–based software reliability model, in: *Proc. Pacific Rim Int'l Symp. on Dependable Computing*, 1999, pp. 143-150.

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

[57] M.Xie, *Software Reliability Modelling*, World Scientific Publishing Company, 1991.

[58] M.Xie, C.Wohlin, An additive reliability model for the analysis of modular software failure data, in: *Proc. 6th Int'l Symp. Software Reliability Engineering (ISSRE'95)*, 1995, pp. 188-194.

[59] S.Yacoub, B.Cukic, H.Ammar, Scenario–based reliability analysis of component–based software, in: *Proc. 10th Int'l Symp. Software Reliability Engineering (ISSRE'99)*, 1999, pp. 22-31.

[60] `http://www.gnu.org/manual/gprof-2.9.1/html_mono/gprof.html`

[61] `http://xsuds.argreenhouse.com`

[62] `http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V40F_HTML/APS30ETE/TITLE.HTM`