ARCHITECTURE BASED SOFTWARE RELIABILITY

KATERINA GOŠEVA-POPSTOJANOVA and KISHOR TRIVEDI
Department of Electrical and Computer Engineering, Duke University,
Durham, NC 27708-0291, USA
E-mail: {katerina, kst}@ee.duke.edu

Abstract– With the growing emphasis on reuse, software development process moves toward component-based software design. In this paper we present an overview of the architecture-based approach to reliability estimation of the software composed of components. 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. Also, 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.

Keywords– 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. However, a great deal of this research effort has been focused on modeling the reliability growth during the debugging phase [1], [2], [3]. These so called black-box models treat the software as a monolithic whole, considering only its interactions with the external environment, without an attempt to model the internal structure. Their main common feature is the assumption of some parametric model of the 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.

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 software reliability growth models are clearly inappropriate to model such a large component-based software system. Instead, there is a need for white-box modeling approach to software reliability that takes into account the information about the architecture of the software made out of components. The motivation for the use of architecture-based approach for software reliability modeling includes the following:

- developing a technique to analyze the reliability of applications built from reusable and COTS software components
- understanding how the system reliability depends on the component reliabilities 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.

The aim of this paper is to provide an overview of the architecture-based reliability models of the sequential software. 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 discussion of the main assumptions underlying the models, their limitations and applicability is presented in Section 6. A concluding remarks are presented in Section 7.

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 of modularization, a module is conceived as a logically independent component of the system which performs a particular 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. Modules and components will be used interchangeably in this paper.

Architecture of the software. The software behavior with the respect to the manner in which different modules of software interact is defined through the software architecture. Interaction occurs only by execution control transfer and, 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.

Control flow graphs are the classic method of revealing the structure, decision points, and branches in program code. Software architecture adapts the control flow graph principles, thus representing the interaction between modules and possible execution paths; a node i represents a program module, and a directed edge from node i to node j represents a possible transfer of control from i to j.

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.

3. STATE-BASED MODELS

This class of models uses the program flow graph to represent the architecture of the system assuming that the transfer of control between modules has a Markov property. This 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 irreducible and absorbing, where the former represents an infinitely running applications, and the letter a terminating one.

As suggested in [4], 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.

Littlewood model [5]

This is one of the earliest, yet a fairly general architecture-based software reliability model.

Architecture. It is assumed that software architecture can be described by an irreducible SMP, thus generalizing the previous work [6] which describes software architecture with CTMC. 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 $F_{ij}(t)$ with a mean sojourn time μ^{ij}.

Failure behavior. Individual modules, when they are executing, fail with constant failure rates ν_i. The transfer of control between modules (interfaces) are themselves subject to failure; when module i calls module j there is a probability λ_{ij} of a failure's occurring.

Solution method. The interest is focused on the total number of failures of the integrated program 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

$$\sum_i a_i \nu_i + \sum_{i,j} b_{ij} \lambda_{ij}$$

where a_i represents the proportion of time spent in module i, and b_{ij} is the frequency of transfer of control between i and j. These terms depend on $p_{ij}, \nu_i, \lambda_{ij}$, μ^{ij}, and the steady state probabilities of the embedded Markov chain π_i.

Cheung model [7]

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 has a single entry and a single exit node, and that the transfer of control among modules can be described by DTMC with a transition probability matrix $P = [p_{ij}]$.

Failure behavior. The reliability of a module i is R_i.

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 $(1 - R_i)$.

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 transi-

tions. 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 $S = \sum_{k=0}^{\infty} Q^k = (I - Q)^{-1}$, so it follows that the overall system reliability can be computed as $R = S(1, n)\, R_n$.

Laprie model [8]

This model is a special case of Littlewood model and the result, although obtained in a different way, agrees with those given in [5].

Architecture. The software system is made up of n components and the transfer of control between components is described by CTMC. The parameters are the mean execution time of a component i given by $1/\gamma_i$ and the probability q_{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 is such that $b_{ii} = -(\gamma_i + \lambda_i)$ and $b_{ij} = q_{ij}\gamma_i, \quad \text{for } i \neq j$. The matrix B can be seen as the sum of two generator matrices such that the execution process is governed by B' whose diagonal entries are equal to $-\gamma_i$ and its offdiagonal entries to $q_{ij}\gamma_i$, and the failure process is governed by B'' whose diagonal entries are equal to $-\lambda_i$ and offdiagonal entries are zero.

It is assumed that the failure rates are much smaller than the execution rates, that is, the execution process converges towards steady state before a failure is likely to occur. As a consequence, the system failure rate becomes $\lambda_{eq} \sim \sum_{i=1}^{n} \pi_i \lambda_i$, where the steady state probability vector $\pi = [\pi_i]$ is the solution of $\pi B' = 0$. This result has a simple physical interpretation having in mind that π_i is the proportion of time spent in state i when no failure occurs.

Kubat model [9]

This model considers the case of a software composed of M modules designed for K different tasks. Each task may require several modules and the same module can be used for different tasks.

Architecture. Transition between modules follow a DTMC 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 SMP.

Failure model. The failure intensity of a module i is α_i.

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

$$R_i(k) = \int_0^{\infty} e^{-\alpha_i t} g_i(k, t)\, dt.$$

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

$$a_i(k) = q_i(k) + \sum_{j=1}^{M} a_j(k)\, p_{ji}(k).$$

The probability that there will be no failure when running for task k is given by

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

and the system failure rate becomes $\lambda_s = \sum_{k=1}^{K} r_k\, [1 - R(k)]$, where r_k is the arrival rate of task k.

Gokhale et.al. model [10]

The novelty of this work lies in the attempt to determine software architecture and component reliabilities experimentally by testing the application.

Architecture. The terminating application is described by an absorbing DTMC. The trace data produced by the coverage analysis tool called ATAC [11] during the testing is used to determine the architecture of application and compute the branching probabilities p_{ij} between modules. The expected time spent in a module j per visit, denoted by t_j, 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 $\lambda_j(t)$ determined by block coverage measurements during the testing of the application.

Solution method. The expected number of visits to state j, denoted by V_j, is computed by

$$V_j = \pi_j(0) + \sum_{i=1}^{n} V_i \, p_{ij}$$

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

The reliability of a module j, given time-dependent failure intensity $\lambda_j(t)$ and the total expected time spent in the module per execution $V_j t_j$, is given by

$$R_j = e^{-\int_0^{V_j t_j} \lambda_j(t)\, dt}$$

and the reliability of the overall application becomes $R = \prod_{j=1}^{n} R_j$.

Ledoux model [12]

This recent work proposes an extension of the Littlewood model [6] 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 C is modeled by an irreducible CTMC 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 [6], does not affect the software because the execution is assumed to be restarted instantaneously when and where 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 μ_i. When control is transferred between two components then a primary (secondary) interface failure occurs with probability λ_{ij} (υ_{ij}).

Solution method. Following the occurrence of a primary failure recovery state is occupied, and the delay of the execution break is a random variable with a phase type distribution. Denoting by R the set of recovery states, the state space becomes $C \cup R$ and the CTMC is replaced by the process that models alternation operational-recovery periods with a generator that defines transition rate from c_i to c_j with no failure, transition rate from c_i to c_j with a secondary failure, transition rate from c_i to c_j with a primary failure, transition rate from recovery state i to recovery state j, and transition rate from recovery state i to c_j. Based on the model described above the following measures are derived: 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 [13]

This work demonstrates the flexibility offered by discrete event simulation to analyze component based applications. One of the presented case studies models a terminating application, whereas the other one deals with a real-time application with feedback control.

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 [14]
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. This 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 $n_f = Nf_1q_1 + Nf_2q_2 + \ldots + Nf_iq_i$, where Nf_i is the total number of traversals of path i. The system probability of failure on any test run is given by

$$q_0 = \lim_{N\to\infty} \frac{n_f}{N} = \sum_{j=1}^{i} f_j q_j.$$

Krishnamurthy and Mathur model [15]
This method 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. Components and their interfaces are identified, and a sequence of components along different paths is observed using the component traces collected during the testing.

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

Method of analysis. The component trace of a program P for a given test case t, denoted by $M(P,t)$, is the sequence of components m executed when P is executed against t. The reliability of a path in P traversed when P is executed on test case $t \in T$ is given by

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

under the assumption that individual components along the path fail independently of each other. The reliability estimate of a program with respect to a test set T is

$$R = \frac{\sum_{\forall t \in T} R_t}{|T|}.$$

An interesting case occurs when most paths executed have components within loops and these loops are traversed a sufficiently large number of times. Then if intra-component dependency is ignored individual path reliabilities are likely to become low, resulting in low system reliability estimates. In this work intra-component dependency is modeled by "collapsing" multiple occurrences of a component on an execution path into k occurrences, where $k > 0$ is referred as the degree of independence. However, it is not clear how one should determine a suitable value of k.

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 m times with a fixed execution time spent in the component per visit relies on the time dependent failure intensity of a component.

Yacoub, Cukic and Ammar model [17]
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.

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 EC_i. The transition probability PT_{ij} is associated with each directed edge that models the transition from node n_i to n_j. CDG has two additional nodes, start node and termination node.

Failure behavior. The failure process considers component reliabilities RC_i and transition reliabilities RT_{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 latest 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. The models are called additive 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 [18]
This model considers a software system composed of n components which may have been developed in parallel and tested independently. If the component reliabilities are modeled by NHPP with failure intensity $\lambda_i(t)$ then the system failure intensity is $\lambda_s(t) = \lambda_1(t) + \lambda_2(t) + \ldots + \lambda_n(t)$, and the expected cumulative number of system failures by time t is given by

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

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 that case, the time has to be adjusted appropriately to consider different starting points for different components.

Everett model [19]
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. Thus, this approach requires to keep track of the cumulative amount of processing time spent in each component.

When the underlying EET models for the components are NHPP 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. ASSUMPTIONS, LIMITATIONS AND APPLICABILITY

The benefit of the architecture-based approach to software reliability modeling is evident in the context of software system that is a heterogeneous mixture of newly developed, reused and COTS components. However, this approach appears to add complexity to the models and to the data collection as well. Many questions related to the architecture based approach to software reliability modeling 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.

Estimation of individual component reliabilities

Most of the papers on architecture-based reliability estimation assume that component reliabilities are available, that is, they ignore the issue of how to determine component reliabilities. 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, Gokhale et.al. [10] 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 [19] identified guidelines for estimating reliability of the newly developed components by identifying the component's static and dynamic properties and characterizing how usage stresses each component.

However, using arguments from [2] it is clear that it is not always possible to use software reliability growth models for estimating the individual component's reliability. Several other techniques have been proposed. Krishnamurthy et.al. [15] used the method of seeding faults for estimating component's reliability. Voas [20] examined a method to determine the quality of COTS components using black box component testing and system-level fault injection methods.

Estimation of interface reliabilities

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 its reliability. However, when an interface consists of items such as global variables, parameters, and files, it is not clear how to estimate the reliability. Some explanation and analysis about the interfaces between components has been performed by Voas et.al [21]. Also, method for integration testing proposed by Delamaro et.al. [22] seems promising for estimating interface reliabilities.

Validity of Markov assumption

This assumption is used in all state-space models. However, it remains to justify the embedded Markov chain assumption which implies that the next module to be executed will depend probabilistically on the present module only and is independent of the past history.

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 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 [15].

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 [15] and [16].

Extracting the 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. Next a brief description of the tools that can be used to extract the architecture of applications is presented.

The GNU profiler *gprof* [23] 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 [11] is a coverage testing tool that is part of the Software Visualization and Analysis Toolsuite (χSuds) 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 [10].

The ATOM toolkit is part of the Compaq Tru64 Unix (formerly Digital Unix) [24] operating system. It consists of a programmable instrumentation tool and several packaged tools. Prepackaged tools that are useful for extracting the software architecture include those for 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 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 would 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.

7. CONCLUSION

In this paper, we have presented the overview of the architecture-based approach to software reliability modeling. 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 analytically the software reliability by combining the architecture with failure behavior. The models from the path-based class compute the software reliability considering the possible execution paths of the program either experimentally by testing or algorithmically. Finally, 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 assuming that component's reliability can be modeled by a non-homogeneous Poisson process and using the component's failure data.

The underlying assumptions of the architecture-based reliability models are discussed in order to provide an insight into the usefulness and limitations of such models which should be helpful in determining the directions for the future research. These include the level of decomposition, estimation of the component and interface reliabilities, validity of the Markov assumption, considering failure dependencies among components and interfaces, and extracting the software architecture.

ACKNOWLEDGMENT

We acknowledge the financial support of NSF under the grant EEC-9714965 (IUCRC TIE grant between Duke CACC and Purdue SERC). Thanks are also due to Srinivasan Ramani for his help with this paper.

References

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

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

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

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

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

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

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

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

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

[10] 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)*, 13-22 (1998).

[11] 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*, 2-10 (1992).

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

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

[14] M.Shooman, Structural models for software reliability prediction, in *Proc. 2nd Int'l Conf. Software Engineering*, 268-280 (1976).

[15] 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*, 146-155 (1997).

[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)*, 86-89 (1998).

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

[18] 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)*, 188-194 (1995).

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

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

[21] 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)*, 126-135 (1996).

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

[23] *http://www.gnu.org/manual/gprof-2.9.1/html_mono/gprof.html*

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