Architecture-Based Software Reliability Analysis: Overview and Limitations

Swapna S. Gokhale, *Senior Member*, *IEEE*

Abstract—With the growing size and complexity of software applications, research in the area of architecture-based software reliability analysis has gained prominence. The purpose of this paper is to provide an overview of the existing research in this area, critically examine its limitations, and suggest ways to address the identified limitations.

Index Terms—Software reliability, software architecture.

1 Introduction

Reliability analysis of a software application based on its architecture offers many advantages, namely, it enables us to:

1. relate application reliability to its architecture and individual component[1] reliabilities,
2. analyze the sensitivity of the application reliability to the component reliabilities and application architecture,
3. explore architectural alternatives to optimize various application attributes such as performance, reliability, and cost,
4. identify reliability bottlenecks,
5. assess application reliability earlier in the life cycle where maximum latitude exists to take corrective action if the reliability does not meet the desired expectations [14], and
6. assess the reliability of an operational application to identify components that provide the highest potential for reliability improvement.

From the mid to late 1990s, architecture-based software reliability analysis has started receiving a great deal of attention [28], [25], [32], [49], [38], [37], [100], as software applications have grown in size and complexity. The objective of this paper is to provide an overview of the state-of-the-art research in the area of architecture-based software reliability analysis. We then describe the shortcomings and the limiting assumptions underlying the prevalent research. These limitations arise because the prevalent techniques:

1. cannot consider many characteristics and aspects that are commonly present in modern software applications (for example, existing techniques cannot consider concurrent execution of components),
2. provide limited analysis capabilities,
3. cannot be used to analyze the reliability of real software applications due to the lack of parameter estimation techniques, and
4. are not validated experimentally.

We also suggest methods that offer the potential to address the identified limitations.

The layout of the paper is organized as follows: Section 2 provides an overview of the existing techniques. Section 3 describes the limitations of the prevalent research. It also describes briefly how the identified limitations could be overcome. Section 4 summarizes the paper.

2 State-of-the-Art: Overview

In this section, we provide an overview of the existing research in the area of architecture-based software reliability analysis.

The primary objective of architecture-based software reliability analysis has been to obtain an estimate of the application reliability based on the component reliabilities and the application architecture. Prevalent architecture-based analysis techniques can be broadly classified into two categories, namely, path-based [49], [64], [99], [86] and state-based [8], [54], [57]. We explain the difference of the techniques belonging to these two categories with the help of an example application, which has a probabilistic control flow graph shown in Fig. 1. This application, first reported in [8], has been used to illustrate the recent research in architecture-based reliability analysis [36], [20]. The application has 10 components: it begins with the execution of component 1 and terminates upon the execution of component 10. $p_{i,j}$ is the probability that the control is transferred to component j upon the completion of component i. The transition probabilities in the control flow graph depend on the operational profile of the application [72]. We let R_i denote the reliability of component i.

In the path-based approaches, several execution paths through the application, where each path starts at the initial component and ends at the final component, are enumerated. The enumeration of paths could be conducted

1. The terms component and module are used interchangeably in this paper.

• The author is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269. E-mail: ssg@engr.uconn,edu.

Manuscript received 14 Sept. 2005; revised 5 May 2006; accepted 19 Sept. 2006; published online 2 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0122-0905.

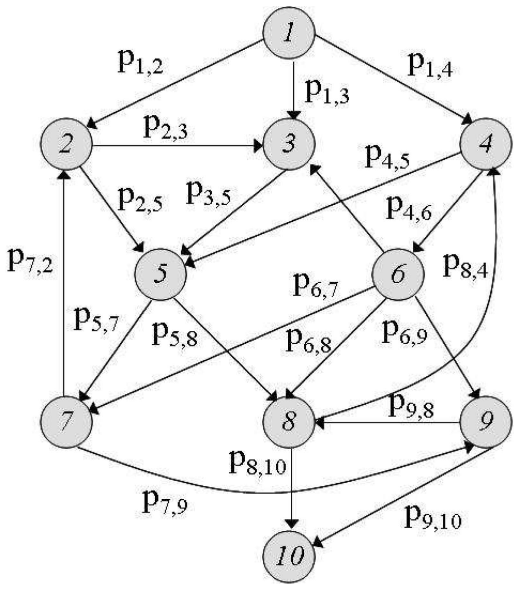

Fig. 1. Probabilistic control flow graph of an example application.

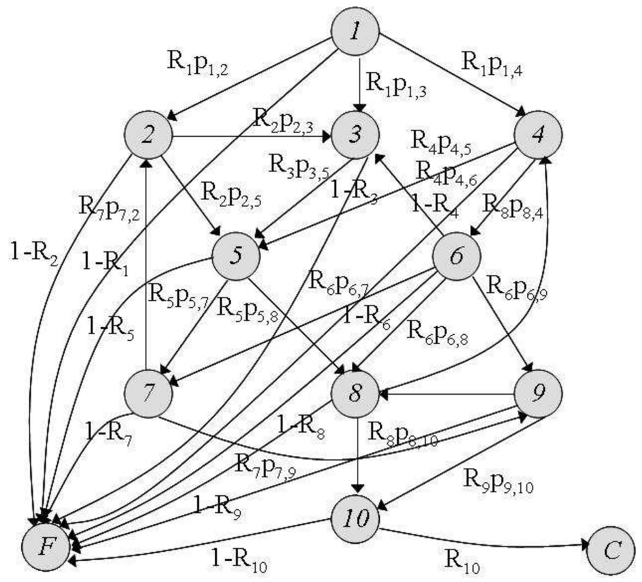

Fig. 2. Composite model of example application in Fig. 1.

algorithmically [100], experimentally [49], or by simulation [25], [42]. The reliability of each path is obtained as a product of the reliabilities of the components along the path. For the example application shown in Fig. 1, 1, 3, 5, 8, 10 is a possible execution path, and its reliability is given by $R_1R_3R_5R_8R_{10}$. An estimate of the application reliability is obtained by averaging the path reliabilities. A notable drawback of path-based approaches is that they provide only an approximate estimate of application reliability when the application architecture has infinite paths due to the presence of loops. For example, in the path 1, 4, 6, 8, $4^{1\ldots*}$, 10, the subpath 4, 6, 8, 4 can occur infinite number of times.

In the state-based approaches, the probabilistic control flow graph of the application is mapped to a state space model. The state space models used to represent application architecture include a discrete-time Markov chain (DTMC) [8], [82], [83], [31], a continuous time Markov chain (CTMC) [53], or a semi-Markov process [58]. Although the path-based approaches represent the failure behavior of the components using the probability of failure or reliability, the state-based approaches allow component failure behavior to be represented using three types of failure models, namely, probability of failure or reliability [8], [31], [82], [83], constant failure rate [53], and time-dependent failure intensity [32]. These failure models can be viewed to form a hierarchy, as far as the level of detail that can be incorporated and the accuracy of the reliability estimate are produced. The reliability estimate is obtained when the component failure model used is the probability of failure or reliability is least accurate since it essentially treats the component as a black box. Representing the component failure behavior by a constant failure rate provides an improvement over the previous case since it can account for the time spent in a component rather than treating every execution of a component as being similar despite the time spent. In general, the more the time spent in a component, the higher is its probability of failure; and using the constant failure rate as the failure model can account for this fact. Representing the failure model by time-dependent failure intensity leads to the most accurate reliability estimate since it can account for the testing characteristics of the component either through the measurement of test coverage [30] or observed failure data. It can also account for dependent executions of a component, for example, the case of a loop [29]. Several combinations of the architectural model and the component failure model are possible, and an overview of these combinations is provided elsewhere [27].

Using the architectural model and component failure models, application reliability may be obtained using two methods in the state-based approaches. In the "composite method," the architectural model and the component failure model are combined to form a single composite model as follows. Two terminal states C and F are added, and these states, respectively, represent the scenarios of successful completion and application failure. For every component i, a directed branch (i, F) is created with transition probability $(1 - R_i)$, representing the occurrence of a failure in the execution of component i. The original transition probability between components i and j is modified to $R_ip_{i,j}$, which represents the transfer of control to component j from component i, conditional to the successful execution of component i. A directed branch is created from the last component to state C with transition probability equal to the reliability of the last component, which represents the correct termination of the last component. The composite model can be solved to obtain the absorption probability in state C, which is the application reliability, using standard methods available for DTMC analysis [89]. For the given values of the parameters of the architectural model and the component failure model, the composite method produces an exact estimate of the application reliability. The composite model of the example application is shown in Fig. 2.

In the "hierarchical method," an estimate of the application reliability is obtained in two steps. In the first step, the model representing the application architecture is solved to obtain the architectural statistics of the application. The architectural statistics include the mean and the variance of the number of visits to each component, and in the case of some models, the expected execution time in each component over a single application run. In the second step, the architectural statistics are combined with the failure parameters of the components to obtain an analytical reliability function.

For the example application in Fig. 1, the analytical reliability function obtained using the hierarchical analysis approach is given by [31]:

$$R = \prod_{i=1}^{10} \left(R_i^{\nu_i} + \frac{1}{2} R_i^{\nu_i} (\log R_i)^2 \sigma_i^2 \right). \quad (1)$$

In (1), ν_i and ${\sigma_i}^2$ denote the mean and the variance of the number of visits to component i and are obtained by solving the DTMC representing the application architecture [47]. Equation (1) is based on a Taylor series expression for the mean of a function of random variable [5]. A second-order Taylor series expression is used because the reliability estimate obtained is an approximation of the exact reliability estimate obtained from the composite approach. The higher the number of terms in the Taylor series expression, the closer the reliability estimate obtained from the hierarchical method to the exact one obtained from the composite method will be.

The hierarchical method described above relies on the assumption of intracomponent independence, which implies that successive executions of the same component are independent of one another. In general, intracomponent independence can lead to a pessimistic reliability estimate [49]. Gokhale and Trivedi [29] represent the failure behavior of a component using time-dependent failure intensity to account for intracomponent dependence. Krishnamurthy and Mathur [49] resolve the issue of intracomponent dependence by collapsing multiple executions of the same component into k occurrences, where k is defined as the degree of dependence.

Fig. 3 shows a pictorial depiction of how the different pieces of information are used for reliability analysis using the composite and the hierarchical methods. Although the composite method provides an exact reliability estimate and the hierarchical method provides only an approximate estimate, the hierarchical method may be preferred due to the following reasons:

- An important use of the architecture-based analysis is in the early phases of the software life cycle, where the sensitivity of the application reliability to individual component reliabilities and to the changes in the application architecture arising due to uncertain or unknown operational profile needs to be determined and different architectural alternatives need to be explored. Sensitivity information can then be used to make decisions such as how many and which components should be developed in-house and which components can be outsourced. To conduct sensitivity analysis using the composite approach, the combined model has to be reconstructed and re-solved, which may be cumbersome and expensive, especially since a software application of moderate size can have thousands of states. On the other hand, the analytical reliability function produced by the hierarchical approach facilitates sensitivity and predictive analysis easily. The analytical reliability function obtained in the hierarchical approach can also be used to propagate the uncertainty in the parameters of the component failure and the architecture models to the application reliability estimate. To demonstrate the applicability of the hierarchical analysis approach for such types of analysis, sensitivity analysis [23] and uncertainty quantification [20] techniques have been developed based on the analytical reliability function in (1).

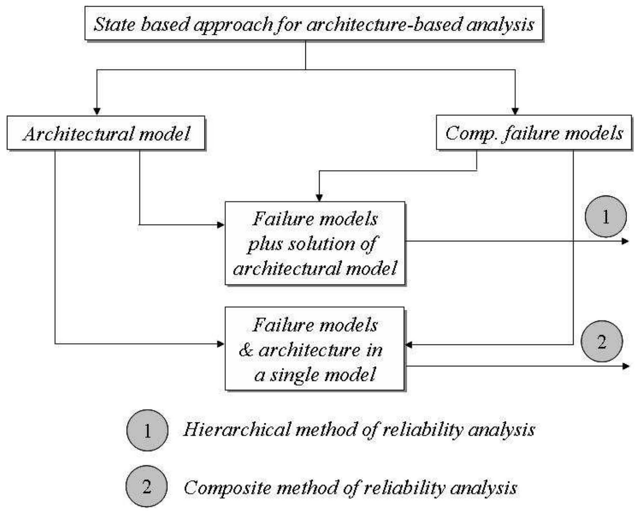
State based approach for architecture-based analysis
Architectural model
Comp. failure models
Failure models plus solution of architectural model
Failure models & architecture in a single model
1
2
1 Hierarchical method of reliability analysis
2 Composite method of reliability analysis

Fig. 3. Analysis methods in state-based approaches.

- The second problem arises due to the stiffness of the composite model. This is due to the fact that the probability of failure of the components may be much lower, compared to the transition probabilities among the components. This difference in the transition and failure probabilities makes transitions to the failure state unlikely. Solution techniques that take into account model stiffness need to be employed in such cases [71].
- Not all combinations of architectural and failure models are tractable analytically using the composite approach. For example, if the architecture of the application is represented by a DTMC and the failure behavior of the components is represented by time-dependent failure intensities, then a composite model based on these two pieces of information cannot be solved analytically or numerically.

The state-based approaches offer several important advantages over the path-based approaches. First, they can consider the impact of infinite paths resulting from the presence of loops analytically. Second, they can consider different types of failure models depending on the level of information that is available. As a result, the state-based approaches can be used through the development life cycle, especially in the testing phase, to determine the impact of reliability growth of an individual component on the overall application reliability. Further, a more accurate reliability estimate may also be obtained by the virtue of incorporating additional information and by considering intracomponent independence using time-dependent failure intensity to represent the failure behavior of the components [29]. Third, the analytical reliability function produced by the hierarchical method in the state-based approaches can enable sensitivity and predictive analysis easily. Of the three types of approaches described above, the state-based approaches have been researched to a greater extent, and this paper is devoted to their discussion.

3 State-of-the-Art: Limitations

In this section, we critically examine the assumptions and discuss the limitations of the prevalent state-based architecture-based analysis techniques. We classify the limitations into five categories, namely, modeling, analysis, parameter estimation, validation, and optimization. We also provide a brief discussion of how the limitations identified in each one of the categories can be addressed.

3.1 Modeling Limitations

In this section, we describe the limitations of the models used for architecture-based analysis. These limitations stem from the assumptions underlying the prevalent models. Some of the assumptions have been made to ensure model tractability. On the other hand, some of the assumptions reflect the characteristics of the applications that existed when architecture-based software reliability research originated, and these assumptions do not hold in the case of modern software applications. Irrespective of the reasons driving the assumptions, they may lead to unrealistic and inaccurate reliability estimates. The limitations of the prevalent architecture-based reliability models include:

1. **Concurrent execution**. State space models assume that at any given instant of time, only one component is executing. As a result, these models cannot represent applications having multiple components executing simultaneously. The assumption of sequential component execution was valid in the context of applications developed using the procedural programming paradigm, but it does not hold for applications developed using the object-oriented paradigm.
2. **Non-Markov transfer of control**. State space models assume that the transfer of control among the components follows first-order Markov property, that is, the next component to be executed depends only on the present component and is independent of the past history. This assumption does not hold for many software applications, where the execution history determines which component is the next one to be executed [97], [95].
3. **Nonexponential sojourn times**. State space models assume that the time spent in each state is exponentially distributed. However, it may be necessary to model the time spent in each state using a general distribution. For example, if a state represents a single execution of a set of instructions or the delay encountered in the transfer of control, then the time spent in that state may be deterministic.
4. **Dependent failures among components**. State space models assume statistical independence among the failures of the components. This independence means that a failure occurring within one component is not the result of a failure occurring within another component, and the failure of one component will not cause any other component to fail. Dependence among components may be a result of data exchanges occurring through parameter or message passing [46]. Popic et al. [76] consider error propagation among the software components using the path-based approach because their technique suffers from the drawbacks of path-based approaches described in Section 2. Dependent component failures may also occur if a set of components share a resource or rely on the same services [39].
5. **Fault detection and repairs during testing**. During the testing phase, it is necessary to determine how fault detection and removal at the component level impacts the application reliability by considering its architecture. A few research efforts have incorporated reliability growth of the components into an architectural model [16], [52]. However, these efforts assume that faults are fixed instantaneously upon detection, an assumption that is easily violated in practice [81]. To the best of our knowledge, presently, no existing model can analyze the impact of reliability growth and finite and imperfect repair, as well as different repair policies [22], at the component level on the application reliability, taking into consideration its architecture.
6. **Alternative configurations during operation**. In order to improve the application reliability during operation in a cost-effective manner, a subset of the application components may be replicated [69]. This subset may be determined based on the component criticalities, which may depend on several factors such as the functionality provided by a component, the execution frequency of a component, and the probability of fault propagation [46]. In such cases, the application reliability needs to be determined, taking into account the component replication and application architecture.
7. **Heterogeneous failure models**. The present state space models require the failure behavior of all the components to be represented by a single type of failure model. In other words, component failure models are required to be homogeneous. However, information at different levels of detail may be available for different components of an application. For example, for components that are developed in-house, information to model their failure behavior using a time-dependent failure intensity may be available, whereas only component reliabilities may be available for the ones that are picked off-the-shelf.

Incorporating additional information for those components for which it is available, even if this information is not available for all the application components, can provide an improvement in the application reliability estimate [29]. However, this will require the capability to represent component failure behaviors simultaneously using different types of failure models or in a heterogeneous manner.

8. **Interface failures**. In the case of modern software applications, interfaces tend to be complex and error prone, which may result in interface failures. Further, the components may be distributed with mechanisms such as RPC [87], Java remote method invocation (Java RMI) [67], common object request broker architecture (Corba) [75], and message-oriented middleware such as Java Message Service (JMS) [68] used to facilitate the distribution. Software components are also distributed for applications that are composed using dynamically discovered components as in the case of composing Web services [39]. For such applications, treating the interfaces as being perfectly reliable may not provide an accurate application reliability estimate. Except for a few research efforts, most of the previous research assumes perfectly reliable interfaces. Cukic [14] considers the failures of the communication link between the hosts in a path-based model. Littlewood [57] uses constant failure rates to represent interface failures in conjunction with the composite solution approach. However, the former suffers from the drawbacks of path-based approaches, whereas the latter suffers from the shortcomings of the composite solution approach.
9. **Architecture styles**. Over the years, recurring patterns of structural organization of components have been identified, and these are referred to as architecture styles. For each style, the constraints imposed need to be represented using a state space model to enable reliability analysis. Adb-Allah [2] identifies the issues involved with using reliability block diagrams to analyze the reliability of architecture styles. Wang et al. [96] describe how some architectural styles may be mapped to state space models for reliability analysis. Only a few simple well-behaved styles are considered in their study, and these include pipeline, batch sequential, and parallel. Models for the reliability analysis of other complex styles, including the pipe and filter style with general topology, event-based, and database (blackboard and repository) styles are currently unavailable. Also, it is unclear how the reliability of an application that follows a heterogeneous style may be analyzed.

We now discuss how the modeling limitations described above can be overcome. To represent the architecture of concurrent applications, a high-level specification mechanism, such as a Stochastic Reward Net (SRN) [77], may be used. Non-Markovian applications may be represented using a higher order state space [55], where the level of history retained in the model needs to be determined empirically. Markov Regenerative Stochastic Petri Nets (MRSPNs) [10] may be used to represent generally distributed and deterministic transitions, whereas Fluid Stochastic Petri Nets (FSPNs) [50] may be used to keep track of the time spent in each component during testing to evaluate the impact of testing and repair of each component on application reliability. Colored Petri Nets (CPNs) [44] may be used to represent the dependencies among components that arise due to error propagation. Interface failures arising due to errors in the interfaces and component distribution may be considered by mapping interfaces to separate states in the model. SRN-based models could also be developed for the reliability analysis of architecture styles. For example, Gokhale and Yacoub [33] develop an SRN model to consider the impact of component failures on the performance of a pipeline software architecture.

3.2 Analysis Limitations

Architecture-based techniques offer the potential of relating the application reliability to the parameters of the component failure models. However, the current arsenal of analysis techniques enables only limited types of analysis. The analysis limitations include:

1. **Reliability estimation**. An important objective of building a model based on the application architecture and component failure behaviors is that the analysis of this model will provide an estimate of the application reliability. Some of the modeling limitations described in Section 3.1 stemmed from the lack of analysis techniques that were required to solve the models that relaxed these assumptions. The lack of analysis techniques was due to the lack of sophisticated numerical methods and computational power when architecture-based analysis research originated.
2. **Sensitivity and importance analysis**. In addition to obtaining an estimate of the application reliability, architecture-based analysis can be used to assess the sensitivity of the application reliability to the component failure parameters. Also, important measures [6] of the components based on the application architecture need to be obtained. Prevalent research, however, addresses the issue of sensitivity analysis only sporadically [59], [99]. To the best of our knowledge, no research has focused on techniques for importance analysis based on the application architecture.
3. **Uncertainty quantification and confidence intervals**. Prevalent architecture-based analysis techniques accept point estimates of architectural and component failure parameters and produce a point estimate of application reliability. These parameters are likely to be uncertain and inaccurate, especially in the early phases. As a result, it is necessary to propagate the uncertainties in these parameters to the uncertainty in the estimate of the application reliability. A few research efforts that address this issue have limited applicability since they consider only a single type of architectural and component failure models [85], [38], [20]. In addition to

uncertainty quantification, it is also necessary to establish a confidence interval for the reliability estimate to determine the risk associated with the estimate.

4. **Prioritization of components**. Application components must be prioritized taking into consideration their sensitivity or importance measures and their contribution to the uncertainty in the application reliability estimate. Prioritization strategies have been developed for series-parallel system architectures [11]. But, these techniques cannot be used for software applications that exhibit complex interactions among the components.

As discussed in Section 2, the hierarchical approach generates an analytical reliability function that can form the basis for sensitivity analysis, uncertainty quantification, and component prioritization. In order to facilitate the hierarchical approach, it is necessary to solve the architectural model to obtain architectural statistics. Packages such as Stochastic Petri Net Package (SPNP) [41] may be used to solve the architectural models represented by SRNs, MRSPNs, and FSPNs numerically. If a numerical solution is infeasible, the architectural model may be simulated to obtain architectural statistics. DesignCPN [45] may be used to simulate a colored Petri net. To enable the use of the hierarchical approach, application reliability in each state needs to be determined and combined with the architectural statistics in order to obtain the analytical reliability function.

During the solution of the architectural models, state space explosion may be an issue, and appropriate model decomposition strategies need to be developed to alleviate this problem. A candidate model decomposition strategy may consist of dividing the application components into groups, obtaining the analytical reliability function for each group, and subsequently aggregating the group reliability functions into an application reliability function. Division of the components into groups may be based on the application and component characteristics. For example, grouping may be based on the functions offered by the application, or according to the sources from which components are obtained (in-house and off-the-shelf). Additional grouping strategies and their relative advantages and disadvantages, along with their impact on the accuracy of the application reliability estimate need to be investigated.

For applications that are composed on the fly such as the composition of Web services, the architecture changes dynamically. The components chosen for the composition may depend on how well they satisfy the expected reliability target (and other nonfunctional attributes including performance), and this needs to be determined based on the application architecture. To facilitate such decision making at runtime, it is necessary to develop methods to specify the application architecture and automatically generate the corresponding reliability model. Efficient methods to solve the generated reliability model to guide runtime decision making are also necessary.

3.3 Parameter Estimation Limitations

In order to enable the use of architecture-based techniques for the reliability analysis of real software applications, it is imperative to be able to estimate the parameters of the architectural and component failure models from different software artifacts. Research in the area of parameter estimation techniques, however, is still in its infancy. Goseva-Popstojanova et al. [38] use fault insertion experiments to estimate the reliability of the components of an application. Gokhale et al. [32] use coverage data collected from the testing of the application to estimate the parameters representing the DTMC model of an application.

In general, profile data generated during the execution of an application may be used to estimate the parameters representing the application architecture. Profile data may include the time spent in code segments, the execution statistics of code segments, execution frequencies of the outcomes of a decision, and resource consumption information. Since profiling involves instrumentation of the software application to generate runtime data, profiling should be sufficiently lightweight so that it does not alter the behavior of the application. Tradeoffs among the overheads incurred in profiling and the granularity of data collection and subsequent analysis enabled must be investigated.

The parameters representing the failure behavior of components may be estimated using statistical testing [38], failure data collected during unit testing of the component using an appropriate Software Reliability Growth Model (SRGM) [17], fault seeding [38], a combination of fault prediction techniques [15], [24], [56], [98] and fault exposure ratio [73], and the testability metric [91]. Code coverage measurements may also be used to estimate the failure parameters of the components [7], [43], [62], [26]. To estimate fault propagation probabilities among the components, a fault injection technique may be used [91], [46].

The techniques described above can be applied to different software artifacts. In the design phase, the description of the software application in a modeling language such as Unified Modeling Language (UML) [12] or a specification language such as Services Description Language (SDL) can serve as the artifact. Real-Time Object Oriented Modeling (ROOM) architecture description language [84] and ObjecTime tool [74] can be used to generate profile data. Metrics-based approaches [88], [101] can be applied to the statechart description of a component to estimate its failure parameters. During the testing and operational phases, the source and/or object code of the components is available. In these phases, profile data may be generated using tools such as *gprof* [18], *quantify* [79], *Java Virtual Machine Profiling Interface (JVMPI)* [66], and coverage analysis using tools such as *Telcordia Software Visualization and Analysis Tool Suite* [9], *purecov* [80], and *Generic Coverage Tool (GCT)* [63].

3.4 Validation Limitations

Ideally, architecture-based reliability analysis should be used in the forward engineering of a software application starting from the early phases of the software development life cycle. However, in order to inspire confidence in the results produced by these techniques in the early phases, when not much information about an application is available, these techniques must be applied, and their results must be validated using software applications, which have been implemented and are operational. Currently, to the best of our knowledge, very little effort has been devoted to the validation of architecture-based reliability analysis techniques.

A number of applications available in the public domain, such as the JPEG library [93], network simulators such as Maryland Routing Simulator (MaRS) [4] and Network Simulator-2 (NS-2) [1], and so forth, could serve as test beds for experimental evaluation. Evaluation using industrial strength applications is also necessary. The results obtained from the evaluation process may be validated using both conceptual and empirical validation [3]. Conceptual validation is qualitative in nature and may be used to validate the relative contribution of each component. It could be performed by consulting with designers, architects, developers, students (in the case of student projects), and in some cases even users. Empirical validation may be used to quantitatively validate the application reliability estimate and the estimates of the contribution of each component. Based on the error logs generated during testing, errors may be seeded into the application, and the application can be retested based on the same operational profile used to generate the profile data. Based on the observed failures and successful executions during retesting, measures similar to the ones in [94] may be defined to obtain an estimate of the application reliability and the contribution of each component. The conceptual approach was used by Gokhale et al. [32], whereas the empirical approach was used by Goseva-Popstojanova et al. [35] for reliability validation of a sequential application with architecture represented by a DTMC.

3.5 Optimization Limitations

Optimization is commonly used in many engineering disciplines to enable the exploration of alternative configurations and to trade off multiple attributes against each other. For a software application, however, optimization based on architecture remains relatively unaddressed. Many exact and heuristic optimization techniques have been developed for series-parallel architectures [51]. These techniques, however, are not useful for a software application since the interactions among its components are more complex than those permitted in series-parallel architectures.

For a software application, the objective of the optimization will depend on the phase of the software life cycle. During the design phase, reliability constrained cost minimization and cost constrained reliability maximization could be the two objectives [40]. During the testing phase, the objective of the optimization may be to determine the allocation of testing effort so that the desired reliability objective is achieved [61], [78]. During the operational phase, optimization may be used to explore alternative configurations and to determine an optimal allocation of components to various nodes in a distributed network to achieve the desired performance and reliability. The existing optimization techniques for software applications consider only a single type of architectural model and a single type of component failure model and can, hence, be used only in that phase of the life cycle during which it is most appropriate to use these models. For example, if the optimization technique is based on reliability as the failure model, then it may be most appropriate to use the technique in the early stages. Also, these techniques assume a continuous relationship between the cost and testing effort expended on a module and its reliability. However, for some components, the cost/reliability relationship may be discrete. For example, in the case of a component that is picked off-the-shelf, only a few cost/reliability combinations may be available for the component. Also, it may be more feasible to predict the range over which component reliability may lie for a given cost based on prior experience rather than predicting the exact reliability value.

Due to the nonlinear and sometimes discrete relationship between the reliability of a component and its cost, it may not be feasible to use conventional nonlinear optimization techniques such as basic descent, gradient projection [60], and modified Newton method [70]. These techniques also have the potential danger of losing the direction of descent, especially when the initial solution is far from the optimal solution [90], [60]. However, by taking advantage of the monotonic relationships that will be satisfied between component (application) reliability and cost and component (application) reliability and performance, the use of heuristic optimization techniques such as simulated annealing [48] and evolutionary algorithms [34], [65] may be explored. Preliminary research in the use of evolutionary algorithms to perform cost/reliability tradeoffs was promising [92], [19], [21].

4 Summary

This paper provided an overview of the state-of-the-art research in the area of architecture-based software reliability analysis, along with an examination of the assumptions and limitations of the existing research. The paper also suggested promising avenues that could be explored to address the identified limitations.

Acknowledgments

This research was supported by the following grants: 1) Large Grant from the University of Connecticut Research Foundation, 2) EPSCoR and Workforce Development Grants from the Connecticut Space Grant College Consortium, and 3) CAREER award from the US National Science Foundation (#CNS-0643971).

References

[1] http://www.isi.edu/nsnam/ns/, 2005.

[2] A. Abd-Allah, "Extending Reliability Block Diagrams to Software Architectures," Technical Report USC-CSE-97-501, Dept. of Computer Science, Univ. Southern California, 1997.

[3] G. Ahrens and A. Chandra, "Availability Modeling and Validation Methodology for RS/6000 Systems," *Proc. Ann. Reliability and Maintainability Symp.,* pp. 305-309, Jan. 1999.

[4] C. Alattinoglu, A.U. Shankar, K. Dussa-Zieger, and I. Matta, "Design and Implementation of MaRS: A Routing Testbed," *J. Internetworking Research and Experience,* vol. 5, no. 1, pp. 17-41, 1994.

[5] L.J. Bain and M. Engelhardt, *Introduction to Probability and Math. Statistics.* Duxbury Press, 1980.

[6] L.W. Birnbaum, "On the Importance of Different Components in a Multi-Component System," *Multivariate Analysis II,* Krisnaiah, ed., Academic Press, 1969.

[7] M.H. Chen, M.R. Lyu, and W.E. Wong, "Effect of Code Coverage on Software Reliability Measurement," *IEEE Trans. Reliability,* vol. 50, no. 2, pp. 165-170, June 2001.

[8] R.C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Software Eng.,* vol. 6, no. 2, pp. 118-125, Mar. 1980.

[9] The χSuds Team, "Mining System Tests to Aid Software Maintenance," *Computer,* vol. 31, no. 7, pp. 64-73, July 1998.

[10] H. Choi, V. Kulkarni, and K.S. Trivedi, "Markov Regenerative Stochastic Petri Net," *Performance Evaluation,* vol. 20, nos. 1-3, pp. 337-357, 1994.

[11] D.W. Coit and T. Jin, "Prioritizing System-Reliability Prediction Improvements," *IEEE Trans. Reliability,* vol. 50, no. 1, pp. 17-25, Mar. 2001.

[12] OMG Corp., http://www.uml.org, 2005.

[13] B. Cuick, H.H. Ammar, and K. Lateef, "Identifying High-Risk Scenarios of Complex Systems Using Input Domain Partitioning," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '98),* pp. 164-173, 1998.

[14] B. Cukic, "The Virtues of Assessing Software Reliability Early," *IEEE Software,* pp. 50-53, May/June 2005.

[15] W.M. Evanco, "Poisson Analysis of Defects for Small Software Components," *J. Systems Software,* vol. 38, no. 1, pp. 27-35, 1997.

[16] W.W. Everett, "Software Component Reliability Analysis," *Proc. Application Specific Software Eng. and Technology,* pp. 204-211, Mar. 1999.

[17] W. Farr, "Software Reliability Modeling Survey," *Handbook of Software Reliability Eng.,* M.R. Lyu, ed., pp. 71-117, McGraw-Hill, 1996.

[18] J. Fenlason and R. Stallman, "Gnu gprof," http://www.gnu.org/manual/gprof-2.9.1/html_mono/gprof.html, 2005.

[19] S. Gokhale, "Cost-Constrained Reliability Maximization of Software Systems," *Proc. Ann. Reliability and Maintainability Symp. (RAMS '04),* pp. 195-200, Jan. 2004.

[20] S. Gokhale, "Quantifying the Variance in Application Reliability," *Proc. Pacific Rim Dependability Conf.,* pp. 113-121, Mar. 2004.

[21] S. Gokhale, "Software Application Design Based on Architecture, Reliability and Cost," *Proc. Int'l Symp. Computers and Comm. (ISCC '04),* vol. 2, pp. 1098-1103, July 2004.

[22] S. Gokhale, "Software Failure Rate and Reliability Incorporating Repair Policies," *Proc. 10th IEEE Int'l Symp. Software Metrics (METRICS '04),* pp. 394-404, Sept. 2004.

[23] S. Gokhale, "Software Reliability Analysis Incorporating Second-Order Architectural Statistics," *Int'l J. Reliability, Quality and Safety Eng.,* vol. 12, no. 3, pp. 267-290, 2005.

[24] S. Gokhale and M.R. Lyu, "Regression Tree Modeling for the Prediction of Software Quality," *Proc. Int'l Symp. Sustainable Agricultural Technologies (ISSAT '97),* pp. 31-36, Mar. 1997.

[25] S. Gokhale, M.R. Lyu, and K.S. Trivedi, "Reliability Simulation of Component-Based Software Systems," *Proc. Ninth Int'l Symp. Software Reliability Eng. (ISSRE '98),* pp. 192-201, Nov. 1998.

[26] S. Gokhale and R. Mullen, "From Test Count to Code Coverage Using the Lognormal Failure Rate," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '04),* pp. 295-395, Nov. 2004.

[27] S. Gokhale and K.S. Trivedi, "Analytical Models for Architecture-Based Software Reliability Prediction: A Unification Framework," *IEEE Trans. Reliability,* accepted for publication.

[28] S. Gokhale and K.S. Trivedi, "Structure-Based Software Reliability Prediction," *Proc. Fifth Int'l Conf. Advanced Computing (ADCOMP '97),* pp. 447-452, Dec. 1997.

[29] S. Gokhale and K.S. Trivedi, "Dependency Characterization in Path-Based Approaches to Architecture-Based Software Reliability Prediction," *Proc. First Application Specific Software Eng. Technology (ASSET '98),* pp. 86-89, Mar. 1998.

[30] S. Gokhale and K.S. Trivedi, "A Time/Structure Based Software Reliability Model," *Annals of Software Eng.,* vol. 8, pp. 85-121, 1999.

[31] S. Gokhale and K.S. Trivedi, "Reliability Prediction and Sensitivity Analysis Based on Software Architecture," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '02),* Nov. 2002.

[32] S. Gokhale, W.E. Wong, K.S. Trivedi, and J.R. Horgan, "An Analytic Approach to Architecture-Based Software Performance and Reliability Prediction," *Performance Evaluation,* vol. 58, no. 4, pp. 391-412, Dec. 2004.

[33] S. Gokhale and S. Yacoub, "Performability Analysis of a Pipeline Software Architecture," *Proc. Int'l Conf. Computer Science and Applications,* July 2005.

[34] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[35] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large Empirical Case Study of Architecture-Based Software Reliability," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 43-52, Nov. 2005.

[36] K. Goseva-Popstojanova and S. Kamavaram, "Assessing Uncertainty in Reliability of Component-Based Software Systems," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 307-320, 2003.

[37] K. Goseva-Popstojanova and S. Kamavaram, "Software Reliability Estimation under Uncertainty: Generalization of the Method of Moments," *Proc. Eighth IEEE Int'l Symp. High Assurance Systems Eng.,* pp. 209-218, 2004.

[38] K. Goseva-Popstojanova, A.P. Mathur, and K.S. Trivedi, "Comparison of Architecture-Based Software Reliability Models," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 22-31, 2001.

[39] V. Grassi, "Architecture-Based Dependability Prediction for Service-Oriented Computing," *Proc. Workshop Architecting Dependable Systems,* 2004.

[40] M.E. Helander, M. Zhao, and N. Ohlsson, "Planning Models for Software Reliability and Cost," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 420-434, June 1998.

[41] C. Hirel, B. Tuffin, and K.S. Trivedi, "SPNP: Stochastic Petri Nets. Version 6.0," *Proc. Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS '00),* 2000.

[42] R. Huang, M.R. Lyu, and K. Kanoun, "Simulation Techniques for Component-Based Software Reliability Modeling with Project Application," *Proc. Int'l Symp. Information Systems and Eng. (ISE '01),* pp. 283-289, 2001.

[43] R. Jacoby and K. Masuzawa, "Test Coverage Dependent Software Reliability Estimation by the HGD Model," *Proc. Third Int'l Symp. Software Reliability Eng.,* 1992.

[44] K. Jensen, *Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Springer, 1997.

[45] K. Jensen, "DesignCPN, Version 4.0," technical report, Univ. Denmark, 1999.

[46] A. Jhumka, M. Hiller, and N. Suri, "Assessing Inter-Modular Error Propagation in Distributed Software," *Proc. 20th IEEE Symp. Reliable Distributed Systems,* Oct. 2001.

[47] J.G. Kemeny and J.L. Snell, *Finite Markov Chains.* D. Van Nostrand, 1960.

[48] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, pp. 671-680, May 1983.

[49] S. Krishnamurthy and A.P. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of Its Components," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 146-155, Nov. 1997.

[50] V. Kulkarni and K.S. Trivedi, "FSPNs: Fluid Stochastic Petri Nets," *Proc. 14th Int'l Conf. Applications and Theory of Petri Nets,* pp. 24-31, 1993.

[51] W. Kuo and V.R. Prasad, "An Annotated Overview of System-Reliability Optimization," *IEEE Trans. Reliability,* vol. 49, no. 2, pp. 176-187, June 2000.

[52] J.C. Laprie, M. Kaaniche, and K. Kanoun, "Modeling Computer Systems Evolutions: Non-Stationary Process and Stochastic Petri Nets—Application to Dependability Growth," *Proc. Sixth Int'l Workshop Petri Nets and Performance Models,* pp. 221-230, Oct. 1995.

[53] J.C. Laprie and K. Kanoun, "X-Ware Reliability and Availability Modeling," *IEEE Trans. Software Eng.,* vol. 15, pp. 130-147, 1992.

[54] J.C. Laprie and K. Kanoun, "Software Reliability and System Reliability," *Handbook of Software Reliability Eng.,* M.R. Lyu, ed., pp. 27-69. McGraw-Hill, 1996.

[55] J. Ledoux and G. Rubino, "A Counting Model for Software Reliability Analysis," *IASTED J. Simulation,* 1997.

[56] M. Lipow, "Number of Faults per Line of Code," *IEEE Trans. Software Eng.,* vol. 8, no. 4, pp. 437-439, July 1982.

[57] B. Littlewood, "A Reliability Model for Markov Structured Software," *Proc. Int'l Conf. Reliable Software,* pp. 204-207, Apr. 1975.

[58] B. Littlewood, "A Semi-Markov Model for Software Reliability with Failure Costs," *Proc. Symp. Computer Software Eng.,* pp. 281-300, Apr. 1976.

[59] J.H. Lo, C.Y. Huang, S.Y. Kuo, and M.R. Lyu, "Sensitivity Analysis of Software Reliability for Component-Based Software Systems," *Proc. 27th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '03),* pp. 500-505, 2003.

[60] D.G. Luenberger, *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1974.

[61] M.R. Lyu, S. Rangarajan, and A.P.A. van Moorsel, "Optimal Resource Allocation of Test Resources for Software Reliability Growth Modeling in Software Development," *IEEE Trans. Reliability,* vol. 51, no. 2, pp. 183-192, June 2002.

[62] Y.K. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich, "Software Reliability Growth with Test Coverage," *IEEE Trans. Reliability,* vol. 51, no. 4, pp. 420-426, Dec. 2002.

[63] B. Marick, "Generic Coverage Tool: User's Guide," technical report, Testing Foundations, 1992.

[64] D. Mason, "Probabilistic Analysis for Component Reliability Composition," *Proc. Fifth ICSE Workshop Component-Based Software Eng. (CBSE '02),* May 2002.

[65] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs,* second extended ed. Springer, 1994.

[66] Sun Microsystems, http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html, 2005.

[67] Sun Microsystems, http://java.sun.com/products/jdk/rmi, 2005.

[68] Sun Microsystems, http://java.sun.com/products/jms, 2005.

[69] M. Casassa Mont, A. Baldwin, Y. Beres, K. Harrison, M. Sadler, and S. Shiu, "Reducing Risks of Widespread Faults and Attacks for Commercial Software Applications: Towards Diversity of Software Components," *Proc. 26th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '02),* pp. 271-276, 2002.

[70] S.H. Mullins, W.W. Charlesworth, and D.C. Anderson, "A New Method for Solving Mixed Sets of Equality and Inequality Constraints," *J. Mechanical Design,* vol. 117, pp. 322-328, June 1995.

[71] J. Muppala, M. Malhotra, and K.S. Trivedi, "Stiffness-Tolerant Methods for Transient Analysis of Stiff Markov Chains," *Microelectronics and Reliability,* vol. 34, no. 11, pp. 1825-1841, 1994.

[72] J.D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software,* vol. 10, no. 2, pp. 14-32, Mar. 1993.

[73] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability—Measurement, Prediction, Application.* McGraw-Hill, 1987.

[74] "ObjecTime User Guide,"technical report, ObjectTime, 1998.

[75] OMG, http://www.omg.org, 2005.

[76] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukie, "Error Propagation in the Reliability Analysis of Component Based Systems," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 53-62, 2005.

[77] A. Puliafito, M. Telek, and K.S. Trivedi, "The Evolution of Stochastic Petri Nets," *Proc. World Congress Systems Simulation,* pp. 3-15, Sept. 1997.

[78] J. Rajgopal and M. Majumdar, "Modular Operational Test Plans for Inferences on Software Reliability Based on a Markov Model," *IEEE Trans. Software Eng.,* vol. 28, no. 4, pp. 358-363, Apr. 2002.

[79] Rational/IBM, http://www.rational.com/products/az/index.jsp, 2005.

[80] Rational/IBM, http://www.rational.com/products/az/index.jsp, 2005.

[81] N.F. Scheidewind, "Fault Correction Profiles," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 257-267, Nov. 2003.

[82] K. Seigrist, "Reliability of Systems with Markov Transfer of Control," *IEEE Trans. Software Eng.,* vol. 14, no. 7, pp. 1049-1053, July 1988.

[83] K. Seigrist, "Reliability of Systems with Markov Transfer of Control, II," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1478-1480, Oct. 1988.

[84] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object Oriented Modeling.* John Wiley & Sons, 1994.

[85] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component-Based Systems," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '01),* pp. 12-21, Nov. 2001.

[86] N.D. Singpurwalla and S.P. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk.* Springer, 1999.

[87] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," Technical Report RFC 1831, Internet Eng. Task Force, Aug. 1995.

[88] M. Tang and M. Chen, "Measuring OO Design Metrics from UML," *Proc. Fifth Int'l Conf. Unified Modeling Language—The Language and Its Applications,* Sept. 2002.

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

[90] A.G. Tsirukis and G.V. Reklaitis, "Application of Generalized Hopfield Networks to Discrete Nonlinear Optimization Problems," *Computers and Chemical Eng.,* vol. 18, pp. 459-468, May 1994.

[91] J.M. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 717-727, Aug. 1992.

[92] S. Wadekar and S. Gokhale, "Exploring Cost/Reliability Tradeoffs in Architectural Alternatives Using a Genetic Algorithm," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '99),* pp. 104-113, Nov. 1999.

[93] G.K. Wallace, "The JPEG Still Picture Compression Standard," *Comm. ACM,* Apr. 1991.

[94] W. Wang, J. Loman, and P. Vassiliou, "Reliability Importance of Components in a Complex System," *Proc. Ann. Reliability and Maintainability Symp.,* Jan. 2004.

[95] W. Wang, D. Pan, and M. Chen, "Heterogeneous Software Reliability Modeling," *Proc. 13th Int'l Symp. Software Reliability Eng.,* pp. 41-52, Nov. 2002.

[96] W. Wang, Y. Wu, and M.H. Chen, "An Architecture-Based Software Reliability Model," *Proc. Pacific Rim Dependability Symp.,* pp. 143-150, Dec. 1999.

[97] D.M. Woit, "Specifying Component Interactions for Modular Reliability Estimation," *Proc. First Int'l Software Quality Week Europe,* Nov. 1997.

[98] Z. Xu, T.M. Khoshgoftaar, and E.B. Allen, "Prediction of Software Faults Using Fuzzy Nonlinear Regression Modeling," *Proc. Fifth Int'l Symp. High Assurance Systems Eng.,* pp. 281-290, Nov. 2000.

[99] S. Yacoub, B. Cukic, and H. Ammar, "Scenario-Based Analysis of Component-Based Software," *Proc. 10th Int'l Symp. Software Reliability Eng.,* Nov. 1999.

[100] S. Yacoub, B. Cukic, and H. Ammar, "A Scenario-Based Analysis for Component-Based Software," *IEEE Trans. Reliability,* vol. 53, no. 4, pp. 465-480, 2004.

[101] W.M. Zage and D.M. Zage, "Evaluating Design Metrics on Large Scale Software," *IEEE Software,* vol. 9, no. 4, pp. 75-81, July 1993.

Swapna S. Gokhale received the BE (Hons.) degree in electrical and electronics engineering and computer science from the Birla Institute of Technology and Science, Pilani, India, in 1994, and the MS and PhD degrees in electrical and computer engineering from Duke University in 1996 and 1998, respectively. She is an assistant professor in the Department of Computer Science and Engineering at the University of Connecticut (UConn). Prior to joining UConn, she was a postgraduate researcher at the University of California, Riverside, and a research scientist in the Applied Research Division at Telcordia Technologies, Morristown, New Jersey. Her research interests lie in the areas of software reliability, software performance, quality-of-service assurance of wireless and wireline networks, and application-level intrusion detection. She is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.