Architecture-Based Software Reliability Estimation: Problem Space, Challenges, and Strategies*

Ivo Krka, Leslie Cheung, George Edwards, Leana Golubchik, and Nenad Medvidovic
Computer Science Department
University of Southern California
{krka, lccheung, gedwards, leana, neno}@usc.edu

Abstract

In modern software-intensive systems, reliability is considered to be one of the most critical non-functional properties. To build software in a cost-efficient manner, reliability should be analyzed at architecture design time. In this paper, we consider the problem space of, challenges in, and strategies for architecture-based estimation of a software system's reliability. Architecture-based reliability estimation is challenging — during early design phases, architects lack information necessary for precisely determining a system's reliability, such as the system's operational profile and failure and recovery information. Thus, we explore how such information can be obtained from alternative sources. Finally, we present a critical overview of existing approaches to architecture-based reliability estimation, in order to indicate directions for future research.

1. Introduction

Ensuring the satisfaction of the non-functional requirements of a software system is as critical as the satisfaction of the functional requirements. Due to the continuing rise of the scale and complexity of software systems, creating a system that actually satisfies such requirements is difficult. Reasoning about non-functional properties cannot be delayed until implementation because the principal design decisions that most heavily impact a system's ability to satisfy non-functional requirements – *i.e.*, the system's architecture – are already deeply incorporated into the realized system. Changing fundamental design decisions after implementation, integration, or deployment is generally prohibitively expensive. For this reason, analysis performed on architectural models is an effective way to reason about the properties of software systems early in the development cycle.

Although a number of techniques for architecture-based reliability estimation have been proposed, they have tended to oversimplify one or more of (1) the definition of reliability of a software system, (2) parameters influencing a system's reliability and (3) the challenge of estimating model parameters by assuming they are available or can easily be obtained. These obstacles must be overcome in order to make reliability estimation on the basis of software architecture meaningful, useful, and usable. Our goal is to encourage future improvement of existing techniques for architecture-based reliability estimation, and to inspire the creation of new techniques, by clearly scoping the problem space, contextualizing and organizing the challenges, and evaluating and critiquing existing approaches.

To define the *problem space* of architecture-based reliability estimation, we must first arrive at a concrete definition of reliability. Reliability is a complex property with different meanings, characteristics, and associated metrics in different contexts, even within a single application or system. This means that it is not possible or desirable to simply provide "the" reliability estimate. Instead, the goal of architecture-based reliability estimation should be to provide the architect with a multidimensional description of reliability from different perspectives, for different parameter combinations, reliability definitions, modes of system usage, and so on. This allows the architects to have a more complete picture of the software system, which help them make better design decisions. Further, architecture-based reliability estimation techniques can point out the sources of system unreliability, and an architect can then employ all this data in making informed design decisions and weighing tradeoffs. We further define and elaborate on the problem space of architecture-based reliability estimation in Section 2.

The primary *challenge* in architecture-based reliability estimation is the lack of precise information about the system, since the system implementation does not yet exist. Existing approaches have fallen short in addressing this challenge, as they focus primarily on modeling the system, ignoring the details of how to gather information to estimate the model's parameters. In Section 3 we provide an extensive list of

* This work has been supported by NSF awards 0417274 and 0720612.

reliability determinants in order to clearly define the complexity of the problem, since many existing approaches work with only a small subset of these ingredients. Architecture-based reliability estimation techniques must be applicable to large-scale, complex systems, for which software architects often lack a complete understanding of (1) the failure behavior, (2) a definitive usage profile, and (3) the behavior and effectiveness of recovery mechanisms. As we describe in Section 3, dealing with these uncertainties requires taking into account multiple system parameters, and estimating or deriving unknown information from indirect or non-ideal sources. Improving the information gathering process would allow an architect to estimate a model's parameters with a higher degree of confidence.

Existing *strategies* for early reliability estimation have strong formal and theoretical foundations. For example, [6] models a software system as a set of interacting components, connections between which represent flow of control, while the approaches of [2], [7], and [10] use UML Sequence diagrams describing main usage scenarios to assess system reliability, and [1] models the reliability of a component based on a representation of its internal behavior. In Section 4, we discuss in detail a set of representative approaches. However, these approaches fall short in various ways when presented with the challenges described in this paper. Firstly, they are not designed to adapt to more complex and nuanced reliability definitions, as described in Section 2. Secondly, almost all of these approaches assume that reliability figures for individual components are available and fixed, no matter the execution environment or the manner in which a component is used. Third, they all assume the existence of operational profiles that fully describe the flow of control between components. Fourth, these techniques oversimplify, and in some cases ignore, numerous other reliability determinants described in Section 3. An additional shortcoming of existing approaches is their lack of scalability: their state space grows rapidly with the size of the system, making execution of the associated estimation algorithms computationally intractable; we expect that development of hierarchical techniques could be an effective way of dealing with these problems.

2. Problem Space

The problem space of architecture-based reliability estimation is directly determined by the specific definition of reliability being considered. For example, in [6], software system reliability is the probability of failure-free operation for a specified time in a specified environment. Although in principle correct, this definition of reliability is oversimplified and incomplete because a number of issues are not addressed. First, this definition implicitly assumes that a concrete definition of a failure exists, and it is not clear how a system failure is defined for an arbitrary software system. Second, the notion of a computational environment is a complex one, and may include a number of different elements, such as the hardware characteristics of the system [5].

Let us examine some possible definitions of failure in different systems. We can say that a system has failed if it produces incorrect results or cannot process requests. Most existing approaches leverage this definition to arrive at a definition of overall system reliability, stating that the system is operational when none of its components have failed. Although convenient, this definition of reliability is not very useful when applied to complex software systems. The definition of reliability for a particular system is more specific and largely depends on the requirements applied to the system.

Not all failures that can occur in a given system have the same weight, meaning that some failures affect a critical functionality of a system, while the impact of other failures is negligible. For example, in an airplane, a failure of a passenger's entertainment console is not a major issue, while the failure of a speed-control component is a safety-critical issue that directly endangers the passengers. At the same time, when analyzing the system, an architect cannot ignore the possibility of a failure within an entertainment console propagating to a critical component.

Even for systems that are functionally very similar, the definition of system failure can be profoundly different. For example, consider two sensor network applications with equivalent architectures, but different uses: the first is used in a medical system, while the other is used in temperature surveillance in a residential building. In the first case, failure of any sensor may constitute a failure of the system, while in the latter case, this is not necessarily so.

Different stakeholders view system reliability from different perspectives. For instance, for a user of a client-server application, a failure of either the client-side or the server-side application results in a system failure. On the other hand, the administrator of the server is likely only concerned with server-side failures. Also, it is possible that in a given system, a system failure can only be correctly defined as a combination of failures of certain components, or as a failure of a certain scenario, complete execution of which is considered critical to the system execution.

As we noted earlier, software-intensive systems should not be observed purely from the point of view of application software because the system will be eventually deployed onto physical hardware hosts, operating systems and middleware platforms in a real-world environment. In such an environment, there are a number of sources of system failures, including hardware host failures, network failures, power source failures, and so on. A definition of reliability must specify which of these will be examined when analyzing system behavior from a certain perspective. Additionally, in a real-world environment, a system can be unreliable even if all software or hardware components are working correctly. For example, in a hard real-time system, if a deadline is not met, the system has failed to perform its task. A similar argument can be made for non-functional requirements other than performance.

Clearly, in most cases it is not possible to simply provide "the" reliability estimate. We view this as a critical shortcoming of the existing research in this area. Defining system reliability is a multi-faceted problem that depends on numerous factors, risks, and uncertainties. Therefore, reliability figures obtained from an architecture-based reliability estimation technique cannot be taken "as is" and applied without further analysis.

3. Information and Challenges

Defining reliability is only the first step in estimating a system's reliability. A lot of additional information is still required. We first elaborate on the required information (Section 3.1) and then pinpoint its subset available at the architecture design-time (Section 3.2). We then discuss other sources from which further required information can be derived (Section 3.3).

3.1. Reliability Ingredients

As discussed above, the first step in determining a system's reliability is deciding on the exact definition of reliability. Closely tied with it is the definition of what constitutes **(1)** *failure-free behavior*, as well as precise notions of **(2)** *failure severity* (*e.g.*, critical vs. minor), **(3)** *failure impact* (*e.g.*, system-wide vs. local), **(4)** *failure extent* (*e.g.*, complete vs. partial), and **(5)** *probability of failure*.

A system's usage in terms of its *operational profile* is another key ingredient of system reliability. Defining an operational profile requires specifying the **(6)** *frequency of execution* of different system services and operations, the frequency and probability of all possible **(7)** *user inputs*, and the **(8)** *operational contexts* in which these processes and inputs occur. The operational profile is necessary in order to estimate the effects of failures.

Once a system fails, it is necessary to know a set of recovery-related parameters: the **(9)** *likelihood of recovery*, **(10)** *time to recovery*, **(11)** *recovery mechanisms* (*e.g.*, redundancy), **(12)** *recovery processes* (*e.g.*, instant vs. incremental), and **(13)** *extent of recovery* (*e.g.*, complete or partial).

Finally, to assess the reliability of a system, we need a model that will allow all of the reliability ingredients enumerated above to be meaningfully combined. In architecture-based reliability estimation we, naturally, use *software architecture models* as an effective basis for reliability analysis. An architectural model specifies the *structure* of the system in terms of components, connectors, and their configurations. An architectural model also captures system *behaviors* in terms of the processing logic of system elements. Furthermore, an architectural model describes the *interaction* between system elements, which includes a precise definition of the mechanisms used for transfer of data and control (*e.g.*, method invocation vs. message-passing) and the protocols and patterns employed (*e.g.*, push- vs. pull-based). Finally, an architectural model expresses *design constraints* that control how architectural elements may be legally composed, utilized, and manipulated.

When all 13 of the parameters enumerated above are known with some certainty, reliability analysis of architectural models can be used with confidence. However, in most cases, information about some of these items is either nonexistent or incomplete. Section 3.2 discusses which items are commonly defined as part of the architectural development process, and can be reasonably assumed to be available, and which are not. In Section 3.3, we suggest how those items that are not commonly available may be approximated or derived from other sources of information.

3.2. Availability of Reliability Ingredients

The information contained in architectural models is rich enough to constitute a backbone for a reliability estimation technique. This section examines how the 13 reliability parameters introduced in the previous section can be obtained from architecture.

Failure information. The definition of (1) failure-free behavior can be directly extracted from an architectural model's specification of desired component and connector behavior. Behavioral specifications within an architectural model must include both functional and non-functional properties to be complete. However, in practice many architectural models do not include complete descriptions of non-functional characteristics, leaving the architect with only a partial definition of failure-free behavior.

The definition of (2) failure severity must be derived and composed from several sources. First, behavioral descriptions in an architectural model may include a specification of the criticality of system services. It directly follows that the severity of a failure of a critical service is higher than that of a non-critical service. Second, user stakeholder perspectives, when captured in an architectural model, help an architect determine failure severity by defining which system users are affected by a given failure.

To determine (3) failure impact of possible system faults, a precise specification of component interactions and deployment is required. Failure impacts are determined by how failures may propagate within the system and whether failures can be contained within a limited scope. For example, if a set of components is deployed within a single process space then failure of one component may (or may not) cause all other collocated components to fail as well.

Information about (4) failure extent can be distilled from architectural specifications that include detailed information about how users interact with the system. Failure extent is determined by the degree to which a given failure impacts the availability of user-level services, or results in degraded operation from the users' perspective. For example, a system may continue to function under a certain failure, but other non-functional properties may be affected (*e.g.*, performance). Conversely, failures that do not impact the experience of system users may be designated as having a minimal failure extent.

The (5) probability of failures cannot, in most cases, be directly extracted from an architectural model. The probability of a given failure may depend on a number of factors which may not be fully determined at architecture design time, such as hardware and network design, or the way in which application logic is implemented.

Operational profile. The (6) frequency of execution of system services is usually not captured in an architectural model. The set of (7) user inputs to the system are normally specified in an architectural model, but the frequencies and probabilities of each possible input are not. This information can, in most cases, be only approximated.

The (8) operational contexts of different system processes can be determined from a compositional evaluation of component behaviors, concurrency mechanisms, computational resources, and so on. For example, an architectural model should specify which services may execute in parallel; this provides information about which other processes may be executing (and consuming computational resources) when a given service is invoked. The availability of computational resources may, in turn, affect a service's reliability.

Recovery information. In principle, most of the failure recovery parameters, which will be revisited below, can be captured in an architectural model. However, they rarely are in practice. The (9) likelihood of recovery cannot be solely determined from an architectural model. For example, an architectural model may specify a mechanism to recover corrupted data (*e.g.* checksum). However, such a mechanism can likely only tolerate certain errors, and an architectural model usually does not specify the probability of one error type versus another as this depends on user inputs.

In general, the (10) time to recover from a failure cannot be ascertained from an architectural model. Firstly, recovering from a failure may require manual intervention, whose duration can be highly variable. Secondly, even when recovery mechanisms are fully automated, the time required to execute may depend on other unknown parameters, such as the services requested by users and the state and availability of computational resources.

The (11) recovery mechanisms employed within a given system may be specified in the system's architectural model. Recovery mechanisms describe the steps taken during normal operation and after failure to ensure that recovery is possible. Recovery mechanisms include strategies like redundancy and replication.

Similarly, a system's (12) recovery processes may be specified in an architectural model. Recovery processes capture the actions performed after failure occurs to mitigate the failure. For example, when a host experiences hardware problems, the system may attempt to automatically redeploy critical components running on that host.

Likewise, an architectural model should clearly specify the (13) extent of recovery in the architectural specification. Certain types of failure cannot be completely recovered from because the software system is not responsible for managing them (*e.g.*, hardware).

3.3. Additional Reliability Information Sources

As described above, to be able to predict the reliability of a software-intensive system, the probabilities of failure and recovery and the operational profile of the system must be known. However, this information is usually not contained in an architectural model. Furthermore, other required information may not be available if the architectural model is incomplete or vaguely defined. This section describes how missing information required for reliability analysis can be derived from various indirect (and possibly imprecise) sources.

Key System Scenarios. The selection and decomposition of key system scenarios – a common step in the requirements elicitation process – provides information relevant to the operational profile by defining important system execution paths. This, in turn, helps an architect to parameterize a reliability model with the probability of different user inputs and processing sequences.

Functionally-similar and predecessor systems. It may be possible to estimate reliability parameters by examining existing systems that are functionally similar. A previous version of a software system could be used to approximate failure and recovery probabilities. One problem with this approach is that the new system will likely be implemented differently, so that its failure parameters may not be directly related to those of any existing system. However, existing systems that provide similar functionality can be used to gather operational profile information.

User information. By gathering information from the intended users of a software system, some reliability parameters related to the operational profile can be estimated. Alternatively, an architect could employ a system "rapid prototype" to gather usage statistics.

Expert knowledge. An architect's previous experience, technical literature, accepted community practices, and industry trends and standards can be invaluable in estimating reliability parameters. For example, within a certain family of applications, some system elements might be known as common sources of faults.

Other models. Models other than software architecture can be an important source of reliability parameters. Detailed hardware models can provide key information about the likelihood of some failures, such as the mean-time-to-failure of a persistent storage drive. Similarly, models of user behavior may shed some light on the system's operational profile.

4. Strategies

Early in this paper, we alluded to some of the technical shortcomings of existing architecture-based reliability estimation approaches. This section provides an overview of the state-of-the-art in the field by characterizing eight approaches that, in our opinion, are representative of the space of existing architecture-based reliability estimation techniques. Specifically, we assess the extent to which existing approaches explicitly account for the reliability ingredients enumerated in Section 3.[2] The approaches discussed include techniques for analyzing reliability at both the component-level [1] and at the system-level [2], [3], [4], [6], [7], [9], [10].

Failure information. Naturally, every reliability estimation technique has a definition of (1) failure-free operation, although some are more simplistic than others. In [6], a failure is defined as the failure of any particular service provided by system components, while in [2], [7], and [10], a failure is defined as the failure of any individual component. [9], on the other hand, allows system failures to be defined as Boolean combinations of individual component failures. None of the described approaches explore the relationship of other non-functional properties to reliability, and we consider the inclusion of these properties within reliability models to be an important direction for future research.

None of the selected approaches, except [1] and [4], explicitly consider (2) failure severity in their analyses. [1] and [4] use multiple failure states to account for failures of different severities. *E.g.*, [4] illustrates how to compute the overall failure severity distribution.

The selected approaches, except [3], do not explicitly include (3) failure impact of possible system faults in their analysis. Rather than considering each failure of a service or component independently from other failures in the system, factors such as error propagation and component collocation must be taken into account in order to determine failure impact. [3] allows architects to specify a probability of propagation when a failure has occurred.

The selected approaches do not differentiate between different (4) failure extents. They consider failure of any part of the component/system as a complete inability to perform further tasks. However, as mentioned earlier, [9] does accommodate one aspect of failure extent by allowing system failures to be defined as Boolean combinations of component failures.

All of these approaches naturally have to use (5) failure probabilities to analyze the reliability, but only approaches [1], [4], and [6] explore their derivation. In [1], the authors apply the method outlined in [8] to identify architectural defects, which are in turn used to identify failures, but they do not specify how to compute failure probabilities. [4] relates the probability of failure to a well-known complexity metric, namely, the number of states and edges in a component's statechart model. [6] derives the failure probability of component services through a combination of the reliabilities of method bodies, method calls and returns, and the environment, but it does not specify how these input values are obtained. Because reliability estimation techniques require failure probabilities as input, methods for deriving failure probabilities from the alternative information sources described in Section 3.3 complement these techniques, and should be developed more fully.

[2] When it is relatively straightforward to extend one of the approaches to account for a reliability ingredient, we acknowledge as much in the discussion.

Failure probabilities are also related to operational profiles, which we describe next.

Operational profile. The (6) frequency of execution of system services and operations is essential for each of the selected reliability estimation techniques. [1] requires the probabilities of transitions between internal component states, while system-level approaches need the probabilities of transfer of control between components and services [6], or the probabilities of execution of particular execution paths [2], [7], [10]. [7] also requires the probabilities of transfer of control between execution paths. However, only in [1] do the authors propose a way of obtaining these parameters: deriving them from a combination of expert knowledge, functionally similar components, and system simulations.

Only approach [2] explicitly provides a modeling notation that accounts for (7) user inputs in reliability estimation, using annotations of UML Use Case diagrams. However, [2] does not specify how the required quantitative data can be obtained. All of the selected approaches could benefit from a technique that provides a description of how users interact with the system by leveraging it to derive operational profiles. As argued in [1], the connection between user inputs and service invocations is not as explicit at the component level as it is at the system level, so component-level reliability estimation techniques have more difficulty incorporating this reliability ingredient in analysis.

The selected reliability estimation techniques pay little or no attention to (8) operational contexts. In [1], [6], and [10], concurrency is not considered. [2] and [7] touch on concurrency within the context of a single scenario, but do not account for the degree of concurrency common in modern software systems (at least not without excessive scalability problems). Additionally, none of these approaches take sharing of computational resources into account, so this represents another important area for further study.

Recovery information. [1] is the only approach that explicitly models the characteristics of failure recovery. The (9) likelihood of recovery and (10) time to recover from a given failure is accounted for in a limited form, as the authors assign a recovery probability to each failure state. We expect that approach in [1] could be further refined to explicitly consider (11) recovery processes, (12) recovery mechanisms, and (13) extent of recovery. For example, [1] can model a partial recovery from a failure by incorporating states that represent degraded operational modes.

The remaining approaches do not explicitly account for recovery, but [6], [7], and [10] can be extended to provide an analysis that includes (9) likelihood of recovery. Similarly, these approaches can be enhanced to include (10) time to recovery; obtaining time to recovery data is a harder problem than incorporating it into the analysis techniques. Most existing approaches also do not consider (11) recovery mechanisms, (12) recovery processes, or (13) extent of recovery. These approaches model systems at a higher-level of granularity than is not ideal for a detailed specification of recovery processes and mechanisms as well as their extent, so while they can, in theory, incorporate these elements, it is difficult to do so in an accurate and informative way.

5. Conclusions

This paper defined the problem space of architectural-based reliability estimation, presented and discussed the reliability ingredients, and described the challenges in obtaining the necessary information for reliability estimation. We have also studied how existing approaches deal with those challenges as well as their shortcomings. Our on-going work is focusing on devising ways to tackle unresolved challenges we have identified. For instance, some of the approaches described suffer from scalability problems when applied to real-world systems. Attempting to address the described challenges in a naive way exacerbates this difficulty.

6. References

[1] Cheung, L., et al., "Early Prediction of Software Component Reliability", in *ICSE 2008*, May 2008.

[2] Cortellessa, V., et al., "Early Reliability Assessment of UML Based Software Models". *WOSP'02*, July 2002.

[3] Cortellessa, V., Grassi, V., "A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems", *CBSE 2007*, July, 2007.

[4] Goseva-Popstojanova K, et al., "Architectural-Level Risk Analysis Using UML," *IEEE Transactions on Software Engineering* ,vol. 29, no. 10, pp. 946-960, October, 2003.

[5] Musa., J.D. "Software Reliability Engineering". McGraw-Hill, 1999.

[6] Reussner R., et al., "Reliability Prediction for Component-based Software Architectures", *J. Systems and Software*, 66(3), 2003.

[7] Rodrigues, G. N., et al., "Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems". In *Fundamental Approaches of Software Engineering (FASE)*, April 2005.

[8] Roshandel R., et al. "Understanding Tradeoffs among Different Architectural Modeling Approaches." *4th Working IEEE/IFIP Conf. on Software Architecture*, June 2004.

[9] R. Roshandel, N. Medvidovic, and L. Golubchik. "A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level." *International Conference on Quality of Software Architectures*, 2007.

[10] Yacoub S.M., et al. "Scenario-Based Reliability Analysis of Component-Based Software." *ISSRE'99*, Boca Raton, November 1999.