Engineering Reliability into Hybrid Systems via Rich Design Models: Recent Results and Current Directions

Somo Banerjee

Nenad Medvidovic

Leslie Cheung

Roshanak Roshandel

Leana Golubchik

Gaurav Sukhatme

1Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{sbannerje,lccheung,leana,neno,gaurav}@usc.edu

2EE-Systems Dept, IMSC
University of Southern California
Los Angeles, CA 90089, USA

3Dept. of Comp. Sci. & Software Engr.
Seattle University
Seattle, WA 98122, USA
roshanak@seattleu.edu

Abstract

Software reliability techniques are aimed at reducing or eliminating failures in software systems. Reliability in software systems has traditionally been measured during or after system implementation. However, software engineering methodology lays stress on doing the “correct things” early on in the software development lifecycle in order to curb development and maintenance costs. In this paper, we argue that reliability of a software system should be assessed throughout the system’s life span, starting with the software architecture level. Our research goal is to estimate the reliability of software systems in early design stages, which we believe involves the ability to reason about numerous uncertainties that exist in this stage, including uncertainty due to lack of execution artifacts. Our proposed approach is to develop techniques that will couple software architectural models with a suite of stochastic reliability estimation models and allow us to reason about these uncertainties. In this paper, we present our recent results using our technique for reliability estimation of software components at the level of software architecture. Another important part of this paper is the discussion of our ongoing research efforts and open research problems in this area.

1. Introduction

1.1. Overview of the Problem

Software reliability techniques are aimed at reducing or eliminating failures in software systems. Existing software reliability techniques are typically rooted in the field of hardware reliability. They complement software testing and generally assume the availability of implementation artifacts. However, the conventional software engineering wisdom suggests that assessing reliability (or any other software quality) at system implementation-time may be too late. If problems are identified at this stage, the system might have to be redesigned and reimplemented, which is overly costly. Many critical design decisions about a system are made long before it is implemented. In this paper, we argue that software reliability should be assessed throughout a system’s life span.

It is widely recognized that the linchpin of the software development process is software architecture [13]. Identifying and mitigating problems early in the software development life-cycle can help to increase the quality of a system in a cost-effective manner. To achieve this goal, we posit that quality attributes must be “built into” the software system during the architectural design phase. However, making useful (quantitative) predictions in early design stages is difficult at best, due to the interplay between many relevant factors, such as complex properties of software components, the potential effects of software on the “firmware” (hardware, OS, device drivers), lack of execution artifacts, and the potentially conflicting desired system attributes. In this paper, we present a technique for component reliability estimation at the software architecture level as a first step of our research on engineering reliability into software systems using rich design models. We present our recent results which indicate the validity and correctness of our approach.

1.2. Motivation

Although several software reliability techniques exist, they are insufficient to estimate reliability at the early design stages for two reasons. First, the existing techniques typically depend on information obtained from the running software system. Second, the existing techniques do not take into account the properties of the firmware underlying the software system which results in reliability estimations “in a vacuum”.

Another challenge to reliability estimation in early design stages is the multiplicity of development scenarios. For example, the “green field” development scenario is one where the entire system is modeled, analyzed, and implemented anew. However, many modern, large software systems are not developed in this fashion. Instead, they involve components that are reused off-the-shelf (OTS), partially captured or outdated models, prototype implementations that require modification, and so on. These are “brown field” development scenarios. Therefore, in a nutshell, the need is to develop a reliability estimation framework that will provide a suite of broadly applicable techniques for assessing the reliability of software-intensive systems as early as practical during their construction.
In this paper, we propose a research direction that will enable an engineer to build a multi-faceted, hierarchical model of a system, tailor it to the development scenario at hand, and assess its reliability in an incremental and scalable fashion. In order to achieve this, we will leverage, and as necessary extend, the concepts from the field of software architecture, which provide high-level abstractions for representing the structure, behavior, and key properties of a software system [13]. A software system’s architecture comprises a set of computational elements (components), their interactions (connectors), and their compositions into systems (configurations).

As part of our ongoing research, we are developing a two-pronged hierarchical approach for modeling and assessing the reliability of software systems. In the first step, we model and assess the reliability at the level of individual components using appropriate stochastic models. The software component models are augmented with the expected firmware properties. Note that a given (reusable) software component may execute in multiple environments, so that it is important to model its software properties independently of the firmware characteristics. At the same time, we need to model the interplay between the software and hardware elements. We are currently investigating hardware properties that directly affect the reliability of a software system. Specifically, we are attempting to identify proper abstractions for describing workload characteristics and demands, properties of a physical host on which components reside (such as over-flow buffers, memory, CPU speed, battery power, and so on), as well as the properties of the communication links between various hosts. The second step in our approach is to compose the individual components’ reliabilities to compute system-wide reliability. We hypothesize that this compositionality has the potential to improve the scalability of our approach.

The contribution of this paper is that we propose a technique that leverages architectural models in generating a stochastic reliability model to estimate reliability of a software component before it has been implemented, i.e., at the architectural level. Another important contribution is that we use our initial results to open up an interesting and challenging research area. We present our current research direction and open research problems in this area, answers to which can result in significant advances to the state of the art in complex systems engineering.

1.3. Related Work

Modeling, estimating, and analyzing software reliability—during testing—is a discipline with over 30 years of history. Many reliability models have been proposed: Software Reliability Growth Models (SRGMs) are used to predict and estimate software reliability using statistical approaches [3,6,9,11]. The common theme across all of these approaches, however, is their applicability to implementation-level artifacts, and reliability estimation during testing. At the architectural level, existing reliability estimation approaches consider only the structure of the system. The only exceptions are [5,18,22,23]. However, none of these approaches consider the effect of a component’s internal behavior on its reliability. They simply assume that the component reliability, or some of its elements (such as reliability of component’s services) is known. They then use these values to obtain system reliability.

If considerable uncertainty in the estimates of the system’s operational profile exists, that uncertainty may manifest itself in the calculated software reliability. Consequently, traditional approaches to software reliability estimation may not be appropriate since they discard any variance due to the uncertainty of the parameters. Some approaches assess the uncertainties in reliability estimation with variable operational profiles via method of moments and Monte Carlo simulation [4,5]; others assume a fixed operational profile and varying component reliability and apply traditional Markov-based sensitivity analysis [1,21].

2. Component Reliability Estimation Framework

Motivation: We recognize that architectural modeling and analysis can provide meaningful insights into a component’s structure and intended behavior and hence can be used as the building block of our reliability estimation technique. Modeling software components from different perspectives can provide complementary views of a component that can lead to sophisticated analyses of its structure, functionality, and non-functional properties. A software component is traditionally modeled from one or more of four functional perspectives: interface, static behavior, dynamic behavior, and interaction protocol [18]. The interface view of a component shows its provided and required services; the static behavior view shows the functionality of the component discretely, i.e., at different “snapshots” during the system’s execution, using invariants on the component states and pre- and post-conditions associated with the components’ operations; the dynamic behavior view shows a continuous view of the component’s internal execution details; and the interaction protocol view shows a continuous external view of a component’s execution by specifying the allowed execution traces of its operations (accessed via interfaces). In this work we leverage these architectural models in two important ways. First, we use the dynamic behavior model as a basis for our estimation technique in the absence of the component’s operational profile. Second, we use view-level inconsistencies between the different architectural models to obtain architectural errors (defects). Defects can then be used to model the failure behavior of the component.

The manner in which these challenges are addressed will depend on the development scenario being used for a partic-
functional component behaviors exhibit the Markov property\(^1\), and another assumption that these approaches have made is that known. We do not believe this assumption to be reasonable. Elements (e.g., reliability of a component’s services) are ability by assuming that component reliability or some of its shows that existing approaches have estimated system reliabilty estimation at the architectural level \([5,18,22]\) currently, this framework hosts three reliability estimation techniques, using Markov Chains (MC), Hidden Markov Models (HMM) and Bayesian Networks (BN). In this paper, we shall elaborate on the reliability estimation technique using HMMs, which has been our main focus to date. The other two techniques will be discussed briefly in Section 4.

A review of available research in the area of software reliability estimation at the architectural level \([5,18,22]\) shows that existing approaches have estimated system reliability by assuming that component reliability or some of its elements (e.g., reliability of a component’s services) are known. We do not believe this assumption to be reasonable. Another assumption that these approaches have made is that component behaviors exhibit the Markov property\(^1\), and have then used Markov models of component interactions as a basis for estimating system-level reliability. We also rely on this assumption, but use it as the basis of constructing a Markov model of interaction between individual states of a component in order to estimate component reliability. However, we are still faced with the challenge posed by the lack of operational profile, especially in “green field” development scenarios, which shall result in unknown parameters in the above-mentioned Markov model. We consequently argue that a Hidden Markov model (HMM) \([15]\) is a formalism that can estimate hidden/unknown parameters in a model, and hence is a suitable approach to handling the lack of an operational profile. Our proposed reliability estimation technique shall build an HMM based on the architectural models of a component. We shall estimate the unknown parameters of this HMM. Then the underlying Markov model of the HMM can be solved to obtain an estimate of component reliability.

**Component Reliability Estimation Technique:** The reliability estimation technique we have constructed as part of this work to date is illustrated in Figure 2 and is an “instantiation” of the Reliability Estimation Framework shown in Figure 1. The technique has three distinctive functional phases. The working example used in this paper is that of a CruiseControl component whose dynamic behavior model is shown in Figure 3. The CruiseControl component has three states, stop \((S_1)\), manual \((S_2)\), and cruise \((S_3)\).

As we have discussed above, HMMs provide an appropriate formalism to compensate for unknown parameters due to the lack of execution artifacts. An HMM is defined by a set of states \(S=\{S_1, S_2, \ldots, S_N\}\), a transition matrix \(A\) representing the probabilities of transitions between states, a set of observations \(O=\{O_1, O_2, \ldots, O_M\}\), and an observation probability matrix \(B = \{b_{ik}\}\) which represents the probability of observing event \(k\) given that we are in state \(i\). We shall now discuss the three functional phases of our technique, relating the discussion to this definition of HMMs.

**Phase 1: Model Extraction.** This phase uses the architectural models of the component to construct an HMM of interaction between individual states of the component. We map the dynamic behavior model to an HMM as follows: the states in the dynamic behavior model become corresponding states in the HMM (set \(S\)), and the event/action pairs of the dynamic behavior model become observations.

---

\(^1\) The Markov property dictates that the probability of transition between two states only depends on the current state, and is independent of the path by which the current state was reached or the amount of time spent in the current state.
of the HMM (set $O$). In order to populate matrix $B$ of the HMM, we need the probabilities of observing various events at each state. These probabilities are hard to obtain at the architectural level due to lack of execution artifacts. We can assume that a domain expert can provide these probabilities, or they can be assigned randomly.

The dynamic behavior model of a component is then extended to add more transitions as described below. Let us illustrate this through an example. Let us consider Figure 4, which illustrates the extensions (i.e., additional transitions, represented by dotted lines) required to the dynamic behavior model of the CruiseControl component. Examples include “eventless” self-transitions with a TRUE label that indicate that the current state of a system may remain unchanged during some time intervals, transitions modeling erroneous behavior (e.g., the car actually stops when gas is invoked), and transitions that occur when an event (e.g., gas) causes an unintended action (e.g., maintain instead of accelerate).

Using this extended behavior model (Figure 4, without state F and its transitions which will be described below), we assume either a random instantiation, or an instantiation of transition probabilities (i.e., frequencies of interactions) by a domain expert. These initial transition probabilities (ITPs) become the initial elements of matrix $A$ of the HMM.

Additionally, this phase also consists of identifying and classifying architectural errors or defects. Our multi-view approach to architecture-level modeling of the component can result in architectural view inconsistencies that can be leveraged to represent defects. Examples of such defects can be a signature mismatch for the gas interface, or a mismatch at the level of static behavior of the cruise interface. We leverage our previous work on defect classification [19] to model the component’s failure behavior. The HMM described above is extended with a failure state ($F$), and transitions from each state in the model to the failure state. As depicted in the figure, a recovery transition is added to designate the recovery state of the component. This state is designated to be the active state of the component once it recovers from a failure. We note here that the technique can be extended to handle multiple failure states, each dealing with errors of different causes, severities, and categories.

**Phase 2: Parameter Estimation.** The output of Phase 1 is an HMM with initial parameters. In Phase 2, we make the HMM “learn” the values of these parameters from a set of synthesized training data using the Baum-Welch algorithm [15]. Currently, we assume that the domain expert is able to provide training data (e.g., via model simulation) to be used to train the HMM. Part of our future work is to investigate suitable techniques for synthesizing training data.

Once the HMM is solved and the unknown parameters of the model are estimated, we leverage a defect quantification method to incorporate the failure behavior of the component into these parameters. The defect quantification technique uses our defect classification in conjunction with a cost framework to quantify the architectural defects obtained from analysis. The cost framework uses a cost function to quantify defects based on the observations that different defects affect a component’s reliability differently, and that the cost associated with defect mitigation varies based on the type of defect. Given the cost of a defect and the frequency of a transition corresponding to it, we can estimate the unknown parameters associated with failure behavior. This can also be used to calculate the reliabilities of individual states of the component. In this phase, therefore, we estimate the unknown parameters of the model and hence obtain the final values of HMM matrices $A$ and $B$.

It is important to note here that defect quantification and training data generation are both “pluggable” components for our technique, i.e., the method used for either of these two computations is independent of the technique itself.

**Phase 3: Reliability Computation.** Upon solving the HMM and estimating its unknown parameters, a Markov model is obtained. Standard techniques involving numerical methods are applied to estimate the reliability of the component, in terms of the probability that it would not be in a failure state over time.

3. Evaluation

Our preliminary results indicate that our model is suitable for identifying trends when estimating reliabilities of software components (e.g., comparing the reliability of various components). This technique is especially suited to “green field” development scenarios where the operational profile of a component may be unknown or inaccurate, or there might not be a one-to-one correspondence between the states of the component and the events triggering the transitions, since the architectural specifications of the compo-
component may be arbitrarily complex. It is also capable of offering analyses aimed at identifying the impact of various factors on a component's reliability. Therefore, we aim to evaluate our technique along two main dimensions: (1) validation against actual component reliability measurements, and (2) usefulness of the analyses enabled by the approach. We realize that the full validation of the approach is a longer term goal and requires availability of real-world systems with architectural models that are verifiably faithful to system implementations. We have thus focused on the second dimension—analyses enabled by our approach—and provide a couple of examples to illustrate its use.

Example 1. We want to identify how a particular state affects the overall component reliability. While Markov-based modeling of the system enables just such analysis, the integration with our defect classification and cost framework enables us to not only identify the critical components, but also identify the operations that have the largest influence on the component's reliability. Such an analysis can be particularly useful for a component with a large number of states. Improving the reliabilities of the critical states can result in improvement in the reliability of the whole component. Figure 4(a) shows the reliability of the individual states of the CruiseControl component ($R_I$ being the reliability of state $S_I$ and so on) and the overall component reliability in three different cases. It is clear from the figure that states $S_2$ and state $S_3$ are more critical than state $S_1$.

Example 2. We want to identify how variations in recovery probability affect the overall component reliability. In a simple experiment, we varied the recovery probability of the CruiseControl component from 0 to 1, keeping everything else constant. Figure 4(b) plots the obtained component reliability vs. recovery probability. This analysis is directly relevant to the design of fault-tolerant components: the impact of individual defects on overall component reliability would be critical in identifying those defects whose recovery should be addressed first.

4. Current Directions and Open Problems

In this section, we outline our current research directions. We take a look at open research problems that we intend to tackle, and propose solutions.

There are three broad research problems that are opened up by an analysis of our recent results described above. First, we intend to replace a single reliability estimation technique with a suite of techniques that can cater to different software development scenarios as discussed in Section 4.1. Second, we intend to marry a system’s software architectural model with the relevant attributes of the firmware it is running on. This will enable us to have more meaningful reliability estimation that takes into account not only the software system but also its environment. This aspect of our research is discussed in Section 4.2. Third, we intend to develop a suite of techniques to estimate reliability of an entire software system at the architectural level, and use the results of our component reliability estimation as the starting point for this purpose. This is discussed in Section 4.3. Section 4.4 describes our planned evaluation methodology.

4.1. Development Scenarios

Our HMM-based component reliability estimation technique is specific to a development scenario where we have good understanding about the dynamic behavior of a component, and enough information to synthesize training data. However, in estimating component- and system-level reliability, we should tailor our approach such that it shall be applicable to different software development scenarios (recall Section 1.2). An architecture-level software reliability modeling framework should be flexible enough to accommodate multiple such scenarios. Therefore, instead of one reliability estimation technique, we intend to have a suite of techniques. As we have already mentioned in Section 2, we include two other reliability estimation techniques in our proposed Reliability Estimation Framework.

One of them is a BN-based approach to component reliability estimation which allows us to avoid estimating transition probabilities between states. The approach may be applicable very early in the development of a new component, where still very little detail is known about the overall system and the component’s future usage. It may also be
applicable in cases where a reusable component is purchased from a third-party vendor, but it is unclear how the component’s internal architecture will be affected by the intended operational scenario in the new usage context.

In our proposed BN-based technique, we will leverage a dynamic behavior model of a component to form a graph where nodes represent states in the dynamic behavior model, and edges (causal relations) describe interactions between states. Additional nodes are introduced to represent failures. Figure 6 is a simple example of such a graph. Then, we propose to take the failure probability of each state (obtained from the defect quantification framework described above) as input to a standard BN inference algorithm, which will produce a component reliability value.

The advantage of this approach is that it can estimate a component’s reliability with little information about the component. However, the results may be inaccurate since we neither know nor attempt to estimate the transition probabilities between states. Intuitively, we anticipate that this will impact the reliability estimation. We will assess this impact in practice and suggest further improvements to this technique as needed.

The other technique assumes that the transfer of control within a component has the Markov property (recall Section 1.3). Thus the component’s behavior can be modeled using a Discrete Time Markov Chain (DTMC) with a transition probability matrix $P = \{P_{ij}\}$ where $P_{ij}$ represents the probability of change from state $i$ to state $j$ in the component. Note that a DTMC assumes that the transition probability matrix is known. This may not be the case when dealing with an architecture-level model. We can still leverage DTMCs in situations where we have different sources of knowledge about a software component. For example, if a component is (1) reused off-the-shelf or (2) still under development but accompanied by a detailed requirements document and/or acceptance test plan, we hypothesize that it will be possible to approximate the component’s future operational profile, i.e., its transition probability matrix. In the first case, the operational profile can be obtained by executing the component in its approximate runtime environment and observing its behavior. In the second case, we intend to leverage existing tools (e.g., Matlab, StateMate) to build executable models and simulate the component’s expected behavior.

4.2. Firmware Models

In addition to a model of a software system, it is important to understand how that system interacts with its environment, including the hardware resources, network, and operating system (referred to as “firmware” for convenience in this paper). Traditionally, hardware and software have been modeled separately. In particular, software system designers have tended to focus on the software models in isolation, assuming that the hardware will somehow (e.g., through the ingenuity of system implementers) meet the required performance and resource constraints. On the other hand, primarily focusing on the hardware system properties provides no guidelines on how to structure the accompanying software. The major problem with either approach is that it is not possible to analyze and verify the entire system.

As in the case of the above software reliability modeling approaches, a common underlying problem with the hardware/software co-design techniques is their dependency on the knowledge of the system’s execution properties, either in the form of statistics about the running system or the exact operational profile needed to estimate reliability. This dependency makes these techniques unsuitable for the early system design phase when the exact operational profile is unknown and any statistical model that can be constructed would by necessity be less precise. We propose to build a more flexible model capable of handling uncertainties associated with early design and less dependent upon accurate estimates of a system’s operational statistics. Our model will enrich a software system’s architectural specification with a set of hardware system properties, such as number of hardware hosts, bus properties between hosts, maximum bus utilization, required memory, maximum CPU load, operating system version, and so on. In effect, we propose to extend the traditional architecture description language (ADL) constructs (which focus on the static, dynamic, structural, and behavioral aspects of a software system) to address the relevant properties of the firmware in a unified model of the system. The existing analysis and verification techniques accompanying ADLs can then be augmented to verify the unified model of software and hardware.

4.3. System-level Reliability Modeling

One way of estimating system-level reliability is to treat the entire system as a single component, build the composite system-level state machine akin to the one depicted in Figure 7, and apply one of the proposed component-level
techniques to it. Clearly this is not a generally applicable approach because it may be difficult to capture all of the complex interactions between the many states of the many different components in a large system's architecture. Furthermore, tractability would also likely become an issue: we may not be able to solve such large models due to their sheer scale. Perhaps most importantly, the approach would fail to take advantage of the component-based nature of software architectures. Architecture-level changes are often accomplished simply by adding, removing, or replacing one or more entire components. In a monolithic reliability model it would be difficult to trace such changes to the complex resulting FSM. Conversely, it would be difficult to trace any reliability concerns back to the component(s) that caused them. Motivated by these observations we intend to build a compositional and hierarchical system-level reliability model, rather than a "flat" one. This model will rely on our component-level reliability calculations. However, in our evaluations we still intend to use the "flat" model as a way of assessing the precision of our hierarchical approach.

One way to achieve scalability is to introduce a hierarchical structure. Such a structure naturally fits the software architectural distinction between components (and connectors) on the one hand, and their configurations on the other. We propose a two-level hierarchical system model.\(^3\) The lower-level model is based on individual components' models (as discussed above). The higher-level model describes interactions between components via their protocols and is solved using the technique most appropriate to the development scenario. Just like it is possible to use different techniques to estimate the reliabilities of different components in a system, we can also use different techniques at the system-level. Additionally, development scenarios at the two levels need not be the same, and our proposed technique should be flexible enough to adapt to this fact. We have begun investigating the implications of this observation. For example, in a scenario we have recently considered [17], we assume that we have some knowledge about individual components, so we use the HMM-based approach at the component-level. At the system-level, however, we assume that we are building a new system and have limited information on how the system will be used. As a result, the BN-based approach is more appropriate.

The resulting higher-level model may still be quite complex in the case of very complex systems. We can safely elide many of the intra-component details because of the model's compositional nature: the reliabilities of the various higher-level model's states will be already calculated at the model's lower-level. However, building the global interaction model still presents a challenge. The difficulty is rooted in combining interaction protocols of interacting components in a scalable manner. Most of the techniques that deal with interacting statecharts (protocol conformance, protocol subtyping, and so on) do so only with pairs of concurrent state machines. The challenge becomes a lot more daunting when trying to take into account arbitrary numbers of interacting state machines. Much of the existing literature in this area refers to theoretical approaches that have not been implemented. We will investigate further in order to devise more practical approaches to this problem.

4.4. Evaluation Methodology

Our two main evaluation metrics for our research will be (1) tractability and (2) sensitivity. In addition, we will also assess empirically the fit of the different proposed reliability assessment techniques to various development scenarios.

Tractability is a measure of a technique's ability to scale up to large models. We will compare the scalabilities of all three proposed component-level reliability estimation techniques for a large number of components with representative characteristics (number of states in the dynamic behavior model, number of transitions, number of interface elements, and so on). We will do so while taking into account different development scenarios (recall Section 4.1), which will allow us also to assess the relative precision (further discussed below) and applicability limitations of the different approaches.

Sensitivity is a measure of a technique's accuracy and usefulness. Since our proposed techniques are primarily intended for the design time, a direct comparison of reliability numbers predicted by them and those measured at runtime is not meaningful. A more useful measure is one which in some manner reflects a confidence in the prediction and sensitivity to changes in the design parameters. Development of such quantitative metrics is part of our proposed effort. Initial candidates include analogues of confidence intervals used in simulations and higher moments of distributions.

Once the component-level reliability measurement techniques are assessed, we will perform analogous benchmark measurements to assess the proposed hierarchical system-level approach. We will compare the results of models composed from components whose reliabilities are estimated homogeneously (e.g., all using HMMs) to those of models composed from heterogeneous models (e.g., using a combination of HMMs, BNs, and MCs). The goal will be to assess the tractability and sensitivity of our system-level approach to different development scenarios.

5. Conclusions

Our reliability estimation framework attempts to bridge the gap between architectural modeling and analyses, and software reliability measurement. We have focused on specific techniques for estimating component- and system-level reliability as a first step in this direction. We have used the

\(^3\) Note that the component-level model can also be hierarchical. So, it is possible to extend our model to multiple levels.
results of this early work to identify a broadly applicable suite of architecture-level reliability estimation techniques. Our approach is flexible, compositional, and hierarchical. We have also discussed the implications of the early results of our reliability estimation framework, and the fact that these results open a new, broad and challenging area of research. We have discussed several possible directions of future research.

References


