Early Prediction of Software Component Reliability

Leslie Cheung¹, Roshanak Roshandel², Nenad Medvidovic¹, Leana Golubchik¹,³

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

²Dept. of Comp. Sci. & Software Engr.
Seattle University
Seattle, WA 98122, USA
roshanak@seattleu.edu

³EE-Systems Dept, IMSC
University of Southern California
Los Angeles, CA 90089, USA

ABSTRACT
The ability to predict the reliability of a software system early in its development, e.g., during architectural design, can help to improve the system’s quality in a cost-effective manner. Existing architecture-level reliability prediction approaches focus on system-level reliability and assume that the reliabilities of individual components are known. In general, this assumption is unreasonable, making component reliability prediction an important missing ingredient in the current literature. Early prediction of component reliability is a challenging problem because of many uncertainties associated with components under development. In this paper we address these challenges in developing a software component reliability prediction framework. We do this by exploiting architectural models and associated analysis techniques, stochastic modeling approaches, and information sources available early in the development lifecycle. We extensively evaluate our framework to illustrate its utility as an early reliability prediction approach.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification – Reliability.

General Terms
Design, Reliability

Keywords
Modeling, Reliability Prediction, Software Architecture

1. INTRODUCTION
Conventional software engineering wisdom suggests that assessing software quality at system implementation-time will often be too late. Many critical design decisions about a software system are made long before it is implemented. Identification of significant problems during implementation or operation can lead to re-engineering of large parts of the system, which has been shown to be prohibitively costly. Therefore, quality attributes must be “built into” the software system throughout design and development, and particularly during architectural design. In this paper, we focus on one such quality attribute—reliability—which can be defined as the fraction of time that a system operates correctly. (A more formal definition is given later in the paper.)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE’ 08, May 10 - 18, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-079-1/08/05...$5.00.

The above suggests that building reliable software systems requires understanding reliability at the architectural level. Several recent approaches have begun to quantify software reliability at the level of architectural models, or at least in terms of high-level system structure (e.g., [2,3,6,8,10,16,22,23]). All of these efforts focus on system-level reliability prediction. While they acknowledge that individual components’ reliabilities have a significant impact on system reliability, these approaches almost invariably assume that the reliabilities of the individual components in a system are known. The few approaches which do consider component-level reliability [8,16], assume that the reliabilities of a given component’s elements, such as its services, are known.

We do not believe these assumptions to be reasonable: it is unclear how the reliability of a component, or its services, is obtained in these approaches. The reliability would either have to be randomly guessed, supplied by an “oracle”, or the component would have to be implemented and one of the existing code-level reliability estimation techniques applied to it. None of these options is satisfying. Three recent surveys [5,10,11] support this observation.

This paper strives to remedy the shortcomings of previous approaches by proposing a framework for predicting reliability of software components during architectural design. This is intended to be complementary to the existing literature on system-level reliability prediction: the values obtained from our component-level approach can be “plugged into” existing (or future) system-level reliability approaches which require this information. We argue that important challenges in component reliability prediction at architectural design time stem from the many uncertainties present early in development and the lack of necessary information about a system and its components. Devising approaches for dealing with two such uncertainties—lack of a component’s operational profile and its failure information—is part of the contribution of our work.

Specifically, one important parameter which may be unavailable or uncertain during architectural design is a component’s operational profile. An operational profile is unavailable since the component has not yet been implemented, hence it is not obvious how one can reliably predict its actual usage. The lack of information about an operational profile is a significant hurdle for system-level reliability prediction techniques as well. However, we argue that operational profile determination is substantially more challenging at the level of an individual component because: (a) information about a software’s usage is typically more readily available at the granularity level of a system, and (b) components are often designed to be used by multiple systems whose usage profiles will differ.

The lack of operational profile information forces us to devise ways of deriving, combining, and applying other existing sources of information available during architectural design. For example,
(1) system engineers’ intuitions can be combined with (2) simulations of a component’s behavior constructed from the architectural model, and (3) execution logs of functionally similar components (e.g., from a previous version of the system under construction). By leveraging these different information sources, we can produce candidate operational profiles for reliability prediction.

Although the above mentioned uncertainties present significant challenges, the availability of formal software architecture models presents an opportunity which we leverage in this work. Specifically, we leverage a component’s state-based models to generate corresponding stochastic models which, in turn, can be used to predict the component’s reliability. In thus utilizing architectural models we observe that another important ingredient in reliability prediction is information about a component’s potential failure modes. However, since software engineers most often design their systems for correct behavior, failure modes are not typically part of an architectural specification. Thus, to handle uncertainties associated with the lack of failure information, we leverage architectural defect classification and analysis techniques [17,19] to identify inconsistencies within a component’s architectural model.

The key contribution of this paper is a framework for reliability prediction of components at the architectural level, which addresses the uncertainty-based obstacles induced by uncertainty. We identify important parameters of the reliability modeling process and study their effects on component reliability. We overcome the lack of failure mode information by utilizing defect analysis and classification techniques. We overcome the lack of operational profile information by utilizing a variety of other available information sources. For instance, we propose a novel approach to using hidden Markov models (HMMs) in estimating operational profiles of a component. We do this by generating HMM training data in a somewhat non-traditional manner.

We evaluate the effectiveness of our reliability prediction process as a function of different information sources. For instance, our results indicate that expert knowledge alone, on which existing approaches often appear to rely, may lead to inaccurate predictions. A rigorous evaluation process on a large number of software components shows that our framework has a high degree of predictive power and resiliency to changes in the identified parameters. The framework is validated by comparisons to an implementation-level technique, which is used as the “ground truth”.

Our framework can meaningfully assess a component’s reliability even when the information is distributed, sparse, and itself not entirely reliable. For instance, our initial hypothesis—that more information about a component (e.g., actual operational profile and failure behavior, and faithful detailed design model or implementation) will result in more precise reliability predictions—has in fact been borne out in our evaluation. Additionally, our results indicate that less information consistently yields more pessimistic predictions, which we consider to be a desirable trait of the framework.

Lastly, we note that predicting a component’s reliability is an important first step to system reliability prediction. Consequently, the work in this paper is part of a larger project, which strives to incorporate component-level reliability predictions into system-level reliability models (e.g., as in [20]).

The rest of the paper is organized as follows. Section 2 reviews existing research that relates to and motivates our work. Section 3 describes our framework in detail. Evaluation results are presented in Section 4. Finally, Section 5 concludes this paper and presents our current and future research directions.

2. BACKGROUND AND RELATED WORK

Over the past thirty years, many software reliability modeling approaches have been proposed. Directly relevant to our work are those that consider the architecture of a system in reliability prediction (e.g., [2,3,6,8,10,16,22,23]). Most of these approaches, with the exception of [8,16] which we discuss below, assume reliabilities of components to be known, and hence focus on predicting system reliability. Moreover, these approaches (sometimes implicitly) assume that the operational profiles of a system are known. As in the case of individual components, obtaining the overall system’s operational profile at the architecture level is non-trivial. This is recognized in [9], which provides an analytical evaluation of the effects of uncertainty in model parameters on the resulting system reliability.

In a risk analysis technique proposed in [8], a component’s reliability risk is defined as a function of the component’s complexity and severity levels of its failures. The complexity is obtained by counting the number of nodes and transitions in the UML statechart model of the component, and the severity levels are assumed to be known. While it is true that the internal structure of a component contributes to its reliability risk, the component’s dynamic behavior is likely to have a much greater effect on its reliability. This is not considered in [8]. In [16], a component’s reliability is computed as a weighted sum of the reliabilities of its services. The reliabilities of component services are assumed to be known, hence this approach suffers from essentially the same shortcomings as the approaches mentioned earlier.

Three recent surveys [5,10,11] directly corroborate our observation that existing approaches may be unreasonable in assuming the availability of individual components’ reliabilities. For example, [10] observes that “[m]ost of the papers on architecture-based reliability estimation [...] ignore the issue of how [component reliabilities] can be determined”, while [11] suggests that “[t]his could even mean that a separate component reliability analysis method is required to complement the system analysis method”.

We address this gap in existing literature by proposing a rigorous approach to predicting component reliability at the architectural level. In a preliminary effort [18], we argued for the need for a component reliability prediction framework. We also identified a set of open research problems. The work presented in this paper provides such a framework and addresses the open problems.

3. THE FRAMEWORK

A software component is traditionally modeled from one or more of four functional perspectives: interface, static behavior, dynamic behavior, and interaction protocol [19]. The interface of a component models its provided and required services; the static behavior shows the functionality of the component at different “snapshots” during its execution, using invariants on the component states and pre- and post-conditions associated with the component’s operations; the dynamic behavior shows a continuous view of the component’s internal execution details; and the interaction protocol shows a continuous view of the component’s interactions with other components.
provides a continuous external view of a component’s execution by specifying the allowed execution sequences of its operations (accessed via interfaces). We use these models as the starting point in component reliability prediction.

For ease of exposition, we present our framework as a three-phase process depicted in Figure 1. Broadly, our framework leverages architecture-level models of a component to construct a stochastic reliability model for that component. We choose to use a stochastic process as our component reliability model for the following reasons. As in most modeling efforts, many details are abstracted away during the component reliability modeling process. These include details about the operating system, hardware on which the component will run, and so on. It is typical in such cases to model the effects of the abstracted details stochastically. Moreover, it is also typical to use specific stochastic processes, namely Markov chains, to allow for a tractable solution. Thus, in this paper, we use a discrete time Markov chain, as has been done in several existing reliability prediction approaches [2, 16, 22].

Briefly, a discrete time Markov chain is a stochastic process with a set of states $S = \{S_1, S_2, \ldots, S_N\}$ and a transition matrix $P = \{p_{ij}\}$, where $p_{ij}$ is the probability of going from state $S_i$ to state $S_j$. Hence, our technique must be able to determine (1) an appropriate set of states of the Markov model (i.e., the set $S$), and (2) appropriate transition probabilities between these states (i.e., the matrix $P$). We will first give a brief overview of each phase of our framework from Figure 1, and then provide their details in Sections 3.1 - 3.3.

In Phase 1 of our framework, we determine the set $S$ by leveraging architectural models and performing standard analyses. There are two types of states in the set $S$ that need to be determined: states corresponding to a component’s normal behavior and to faulty behavior. States corresponding to normal operation can be obtained directly from existing architectural models. However, failure behavior is rarely explicitly modeled at the architectural level [14]. This is the more difficult part of the state determination process. We address this problem by leveraging a defect analysis and classification technique [17] which identifies inconsistencies in a component’s architectural models and classifies them using a architecture-level defect classification scheme. Both of these are tailorable elements of our approach: other analysis and/or classification techniques can be substituted. Details of the state determination process are described in Section 3.1.

Values of the transition matrix $P$ are determined in Phase 2 of our framework using various sources of available information. Given the states, determination of transition probabilities between these states remains a challenge. A critical difficulty here is the lack of information about the operational profile and failure information of the component. We address this problem by (a) identifying and classifying the utility of information sources available during architectural design and (b) combining the use of such sources with a hidden Markov model (HMM)-based approach we outlined in [18]. The description of information sources typically available at the architecture level and the details of determining transition probabilities are described in Section 3.2.

Once the states and the transition probabilities of the Markov chain reliability model are determined, in Phase 3 of our framework the model is solved to compute a reliability prediction. We compute the reliability of a component by solving for the steady state probability of not being in any failure state. Given that we do not expect the state space of an architecture-level component model to be huge (e.g., we do not expect it to be on the order of a million states), in this work, we simply apply standard numerical techniques [21] to solve the Markov chain model, as detailed in Section 3.3. A number of approaches can be taken to ensure tractability if the state space size is determined to be too big [21]. However, these are outside the scope of this paper.

**Modeling Assumptions.** In addition to modeling normal and failure behavior of a component, as in existing approaches (e.g., [2]), we are also interested in modeling recovery behavior. Hence, we can define reliability as the probability that the component is in a state representing normal behavior. Moreover, we assume that, once a failure occurs, the component transitions to a failure state immediately. We have not considered the effect of error propagation, such as the problem studied in [3], in our framework.

**Running Example.** The example that we will use in this section is that of the Controller component of the SC Rover, a third-party robotic testbed based on NASA JPL’s Mission Data System framework [1]. This testbed contains requirements and architectural documentation as well as a simulated robotic platform. SC Rover is the implemented prototype of a robot that is capable of performing different missions such as wall-following, turning at a given angle, moving a fixed distance in a given direction, and identifying and avoiding obstacles. Here, we focus on the behavior of the robot in a wall-following mission: it should maintain a certain distance from the wall; if it moves too far from or too close to the wall, or encounters an obstacle, it has to turn in an appropriate direction to correct this. As soon as the state of the robot changes, it has to update a database with its new state.

### 3.1. Phase 1: Determining States

We view the states in the set $S$ as being of two types: Behavioral, $B$, are the states related to the intended functionality of the component, and are obtained from architectural models. Failure, $F$, are the states related to the improper behavior of the component, and are obtained from defect analysis of architectural models.
We leverage a component’s dynamic behavior model in order to determine behavioral states (set B) of our model. A dynamic behavior model of a software component is often depicted by a state transition diagram that shows the internal states of the component, the transitions between them, and the event/action pairs that govern these transitions (e.g., as in UML’s statechart diagrams). The dynamic behavior model of the Controller component is illustrated in Figure 2a and consists of six states: Idle (B1), Estimating Sensor Data (B2), Turning Left (B3), Turning Right (B4), Going Straight (B5), and Updating Database (B6). We map the states of the dynamic behavior model directly to the behavioral states of the Markov chain reliability model (Figure 2b).

To determine the failure states (set F) we analyze the architectural models of a component. The multi-view approach to modeling a component outlined above and described in [19] allows for the detection of architectural inconsistencies. Standard techniques for architectural analysis [14] can be adopted to this end. The results of architectural analyses can be leveraged to represent defects, which contribute to the unreliability of the component. Recall that we assume that a defect will immediately manifest itself as a transition from an affected state to a failure state.

Defect d1 is shown on the left side of Figure 3: a Turn event with parameter deg=0 represents the fact that the robot is not turning in the dynamic behavior model, while the same is represented by a Go Straight event in the interaction protocol model. Such discrepancies may not be uncommon if the design of the component’s external interfaces and interactions is separated from the design of its internal behavior (e.g., as argued in [4]). If we implement the component according to this dynamic behavior model, the component may be unable to process actions is separated from the design of its internal behavior (e.g., as in UML’s statechart diagrams). The dynamic behavior model of the Controller component is illustrated in Figure 2a and consists of six states: Idle (B1), Estimating Sensor Data (B2), Turning Left (B3), Turning Right (B4), Going Straight (B5), and Updating Database (B6). We map the states of the dynamic behavior model directly to the behavioral states of the Markov chain reliability model (Figure 2b).

For example, we identified two defects in one version of the Controller component, shown in Figure 3. Defect d1 is shown on the left side of Figure 3: a Turn event with parameter deg=0 represents the fact that the robot is not turning in the dynamic behavior model, while the same is represented by a Go Straight event in the interaction protocol model. Such discrepancies may not be uncommon if the design of the component’s external interfaces and interactions is separated from the design of its internal behavior (e.g., as argued in [4]). If we implement the component according to this dynamic behavior model, the component may be unable to process actions is separated from the design of its internal behavior (e.g., as in UML’s statechart diagrams). The dynamic behavior model of the Controller component is illustrated in Figure 2a and consists of six states: Idle (B1), Estimating Sensor Data (B2), Turning Left (B3), Turning Right (B4), Going Straight (B5), and Updating Database (B6). We map the states of the dynamic behavior model directly to the behavioral states of the Markov chain reliability model (Figure 2b).

We note that defects may be different from each other in terms of factors such as severity, the subsystem impacted by the defect, and time and effort needed to recover from the defect. Based on these differences, defects can be partitioned into different classes, each of which may have different failure and recovery characteristics. For example, it was much easier to uncover why the robot was not moving and required manual restarting as a result of defect d1 than why the robot did not turn appropriately as a result of defect d2.

Based on this observation, we use the architecture-level defect classification scheme presented in [17]. Other defect classification methods may be used instead without affecting our reliability model. In our approach, we designated a failure state to each class of defect and represent the i-th defect class with the failure state Fi. Deciding on how to partition the identified defects into defect classes is part of the modeling process. At one extreme, we could aggregate all the defects into a single defect class; this would result in a single failure state in the reliability model. At the other extreme, we could assign each defect its own defect class; this would result in one failure state per identified defect. Using a single defect class (i.e., a single failure state) to represent all identified defects would give us a simpler model. However, the flexibility to include multiple failure states facilitates construction of richer reliability models, which in turn allows for more sophisticated analyses. In Section 4, we illustrate how this flexibility allows exploration of the effect(s) of an individual defect (or a related group of defects) on a component’s reliability. Based on the defect classification we used [17], the two defects d1 and d2 identified in the Controller component are assigned to two different classes D1 and D2, respectively. Therefore, as depicted in Figure 2b, we designate two failure states F = {F1, F2} to correspond to the identified defects.

### 3.2. Phase 2: Determining Transitions

The transitions in our Markov chain model, corresponding to the elements of the transition probability matrix P, can be viewed as
being of three different types: (1) behavioral; (2) failure; and (3) recovery. Behavioral transitions are between two behavioral states; failure transitions are from a behavioral state to a failure state; and recovery transitions are from a failure state to a behavioral state. The process of determining the probabilities of each transition may be different and depends on the information available to the architect. We identify the following possible information sources.

Domain Knowledge. Information about a component may be obtained from a domain expert. The main difficulty is that such an expert may not be available. Even when an expert is available, this information source is inherently subjective and the information may be inaccurate, either due to the complexity of the component or to unexpected operational profiles of that component.

Requirements Document. The requirements for a given component, or the overall system, will frequently contain the typical use cases for that component. Furthermore, the requirements may be explicit in terms of how a component is to respond to exceptional circumstances such as failures. This information can be leveraged to estimate at least a subset of the above transition probabilities.

Simulation. Simulation of a component’s architectural models [7] has the potential of handling components with complex state spaces because the process can be automated. However, simulation techniques still require information related to a component’s operational profile, which would have to come from other sources.

Functionally Similar Component. If a functionally similar component exists, we can use its runtime behavior to estimate the operational profile of the component under consideration. It is also possible to combine information from multiple functionally similar components. For example, if we are building a word processing component with drawing capabilities, we can leverage runtime information of an existing word processor to explore the behavior corresponding to word processing functionality, and the runtime information of an existing drawing tool to explore the behavior corresponding to drawing functionality.

Several of the above information sources may be available simultaneously. Our approach allows us to use them in a complementary manner and thus mitigate their individual disadvantages.

**Determining Behavioral Transition Probabilities.** Let us define \( q_{ij} \) to be the probability of going from behavioral state \( B_i \) to state \( B_j \). The central question here is how to determine the numerical value of \( q_{ij} \). We address this in the context of information sources described above and use the Controller component for illustration. Since in the Controller component the transitions out of state \( B_2 \) are the more interesting ones, we will use them in our examples.

If domain knowledge is available, we can focus on the subset of possible operational profiles corresponding to the provided domain knowledge. For instance, the expert may suggest that in the Controller example the robot moves straight most of the time. Then, we can eliminate the operational profiles corresponding to high probabilities of turning left and right.

When simulation data of a component’s architectural models or from a functionally similar component is available, we can use it to obtain the behavioral transition probabilities. While a standard Markov-based approach would assume that there is a one-to-one correspondence between observed events in the simulation (or execution logs) and transitions in the model, such correspondence may not exist. This is especially true in the case of a functionally similar component. For example, in the Controller component from Figure 2, when we observe the Turn event, we cannot tell whether a transition occurred to the Turning Left, Turning Right or Going Straight states from the Estimating Sensor Data state.

In our preliminary work [18], we suggested that in such a case we can use hidden Markov models (HMMs) [15] to obtain behavioral transition probabilities. An HMM is defined by a set of states \( S = \{S_1, S_2, \ldots, S_N\} \), a transition matrix \( A = \{a_{ij}\} \) representing the probabilities of transitions between states, a set of observations \( O = \{O_1, O_2, \ldots, O_M\} \), and an observation probability matrix \( E = \{e_{ik}\} \), which represents the probability of observing event \( O_k \) in state \( S_i \).

The set \( S \) of the HMM comes from Phase 1. The event/action pairs of the dynamic behavior model become observations of the HMM (set \( O \)). Matrices \( A \) and \( E \) can be initialized with random values [15] or they may be initialized more intelligently, by utilizing architectural models. Our experience shows that random instantiation results in a slower convergence of the HMM model; the details are beyond the scope of this paper.

The Baum-Welch algorithm [15] is commonly used to train an HMM. An input for this training process is called training data, and consists of sequences of observations. Traditionally, training data for HMMs is obtained by collecting measurements using an already built system in an existing operational environment. However, since we are doing this at the architectural level, we needed to find a novel approach to generate training data. To this end, we utilized the available information sources: a combination of expert advice, system requirements, simulation traces (when simulation of architectural models is available), or execution traces (when a functionally similar component is available). Given an initial HMM constructed as described above, the Baum-Welch algorithm converges on a Markov model that has a high probability of generating the given training data. The underlying Markov model of the HMM, with transition matrix \( A \), obtained after running the Baum-Welch algorithm represents the behavioral transition probabilities for the component, i.e., \( q_{ij} = a_{ij} \) for all \( i \) and \( j \).

We note here that the training data includes no failure or recovery behavior. This enables us to focus on behavioral transition probabilities. We will incorporate failure and recovery behavior next, based on the defect classification we performed in Phase 1.

**Determining Failure and Recovery Probabilities.** We define \( f_{ij} \) to be the probability that a defect of class \( j \) occurs while the component is in state \( B_i \). In other words, in the reliability model, \( f_{ij} \) is the probability of going from a behavioral state \( B_i \) to a failure state \( F_j \). Furthermore, we define \( r_{kl} \) to be the probability that the component enters state \( B_k \) after recovery from a defect of class \( k \). For a given pair of behavioral and failure states, \( B_i \) and \( F_j \), we can determine whether \( f_{ij} \) is non-zero. This would be determined as part of the architectural analysis process, as described in Phase 1. Also, for

---

2. We have assumed that a component will recover from a failure due to one defect before experiencing a failure due to another defect. This assumption is not reasonable in multi-threaded components. We treat such complex components as systems and apply our system-level reliability prediction technique on them [20].
each defect class $D_k$, we can determine (e.g., from a requirements
document or domain expert) what is a reasonable set of states in
which the component can re-start after recovery from failure. In
other words, for each behavioral state $B_i$, we can determine
whether $r_{kl}$ is non-zero. In the Controller component from Figure
2b defects of classes $D_1$ and $D_2$ can occur in states $B_2$ and $B_3$,
respectively. Thus, we add transitions from $B_2$ to $F_1$, and from $B_3$
to $F_2$. In this example, recovery from any failure returns the
component back to state $B_1$. The self-transitions at $F_1$, and $F_2$ represent
the component being in a failure state until recovery is complete.

Knowing which failure ($f_{ij}$) and recovery ($r_{kl}$) transition proba-
bilities are non-zero is not sufficient. To complete the reliability
model, we need to assign specific values to these probabilities.
One approach is to explore the design space, i.e., to vary the failure
and recovery probabilities and observe the resulting effects on the
component’s reliability prediction. We demonstrate this approach
in Section 4. This allows us to explore how sensitive the compo-
nent’s reliability is to each of the defect classes and to the recovery
process from each defect class.

We could take advantage of the available information sources to
reduce the design search space once again. For instance, a domain
expert could indicate how difficult it is to recover from a failure
due to defect class $D_k$. In turn, this would suggest the values ranges
for $r_{kl}$ the reliability modeler should consider.

### 3.3. Phase 3: Computing Reliability

In this phase we compute the component’s reliability by solving the
Markov chain reliability model constructed in Phases 1 and 2.

Let $\pi(i)(t)$ be the probability that a component is in state $i$ at time
$t$, where $i = F_1, F_2, B_1, ... B_N$. As $t$ goes to infinity (i.e., as the
component operates for a long time), these probabilities converge to
a stationary distribution,

$$\hat{\pi} = [\pi(F_1), \pi(F_2), \pi(B_1), ..., \pi(B_N)]$$

which is uniquely determined by the following equations:

$$\sum_{i \in S} \pi(i) = 1$$

$$\hat{\pi} = \hat{\pi}P$$

This system of linear equations can be solved using standard
numerical techniques [21]. The component’s reliability can then be
defined as the probability of not being in a failure state:

$$R = 1 - \sum_{i=1}^{M} \pi(F_i)$$

As an illustration, assume that in our example from Figure 2b the
non-zero failure probabilities are $f_{21} = 0.05$, $f_{32} = 0.04$, and that the
non-zero recovery probabilities are $r_{11} = 0.2$, $r_{21} = 0.8$. These values
were obtained from SCRover’s chief developer (i.e., domain
expert). This gives us the matrix $P$ in (3).

Note that this matrix is relatively sparse. We expect this to be
the case for reliability models of many (though by no means all) real
components, which may provide separate operations, modes of
operation, sub-components, and so on. Thus both storage require-

3. It is not difficult to show that for our reliability model this limiting dis-
tribution exists and is a stationary one [21].

<table>
<thead>
<tr>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.8</td>
<td>0.2</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.05</td>
<td>0.019</td>
<td>0.076</td>
<td>0.095</td>
<td>0.8455</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0.04</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.96</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.96</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8455</td>
</tr>
</tbody>
</table>

ments (for the matrix $P$) and the computation requirements (of
solving Equation (1)) can be improved [21], if needed.

After solving Equation (1), we have

$$\hat{\pi} = [\pi(F_1), \pi(F_2), \pi(S_1), \pi(S_2), \pi(S_3), \pi(S_4), \pi(S_5), \pi(S_6)]$$

$$= [0.0765 0.0012 0.0220 0.3061 0.0233 0.0029 0.2840 0.2840]$$

Thus, the reliability of the Controller component is

$$R = 1 - (0.0765 + 0.0012) = 0.9223$$

In the next section, we discuss how such reliability predictions are
intended to be interpreted and used.

### 4. EVALUATION

In this section, we validate and support several claims we have
made throughout the paper. This includes (a) showing the effect-
iveness of our approach when different sources of information are
available, and (b) showing the predictive power and resiliency to
changes in parameters identified in Section 3. Since our framework
is intended to be used at design-time, a direct comparison of reli-
ability numbers predicted by the framework and those measured at
runtime would not be meaningful. Design-time approaches are
intended for relative comparisons between possible fault mitiga-
tion choices rather than (literally) accurate reliability predictions.
Hence, a more useful measure here is one that in some manner
reflects a confidence in the prediction and sensitivity to changes in
the component and reliability model-related parameters.

In our evaluation, we first compare the sensitivity of our results to
the different information sources (recall Section 3.2). Next, we
show how the estimates of operational profiles affect the predicted
component reliability values. Finally, we study sensitivity of the
results obtained using component models of different granularities.
We have applied our framework in the context of a large number of
components whose architectural models we were able to develop
or obtain. Examples include components from a cruise control sys-
tem [18]; the SCRover robotic testbed [1], developed by a separate
research group at USC in collaboration with NASA’s JPL; MIDAS
[12], a large, embedded system developed as part of a separate col-
laboration between USC and Bosch; DeSi [13], an architectural
design and analysis tool developed as part of a separate research
project at USC; and a large library of systems developed in USC’s
undergraduate software engineering project course [24].

In order to observe the trends in our framework’s reliability predic-
tions on sufficiently large numbers of components with controlled
variations, as part of our evaluation we have also synthesized many
state-based models for “dummy” components, and performed eval-
uations on those models.

Our framework has consistently yielded qualitatively similar
results for all of the above cases. To illustrate these results and
highlight the framework’s key properties, particularly its sensitiv-
ity, we will use as an example a representative component from the DeSi environment [13]. Results from a number of other components we have evaluated are available in [25].

DeSi is an environment that supports specification, manipulation, and visualization of deployment architectures for large distributed systems. It consists of three major subsystems: DeSiModel that stores information about the current deployment; DeSiView that visualizes information in the DeSiModel subsystem; and DeSiController that generates deployment plans based on constraints set by the user, allows users to fine-tune parameters of a generated deployment, and invokes redeployment algorithms [13] that update the DeSiModel. To demonstrate our approach’s ability to handle components of large scale and complexity, we will treat each subsystem as a single component.

DeSi served as a particularly useful evaluation platform because it was designed and implemented from an architecture-centric perspective: it contained clearly identifiable components, which composed hierarchically into DeSi’s subsystems, and was accompanied by existing architectural models. For consistency, we will show the evaluation results of applying our reliability prediction framework to the DeSiController only. A slightly abridged dynamic behavior model of DeSiController is depicted in Figure 4a. To evaluate our framework in a controlled manner, we injected architectural defects into DeSi. Table 1 summarizes the subset of defects used in the results presented in the remainder of this section. We consider each defect in Table 1 as being in a different defect class.

<table>
<thead>
<tr>
<th>Defect</th>
<th>Description</th>
<th>Affected State</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Mismatched signatures</td>
<td>Waiting for command</td>
</tr>
<tr>
<td>d2</td>
<td>Missing model validation rules in the design document</td>
<td>Validating model</td>
</tr>
<tr>
<td>d3</td>
<td>Mismatch between the dynamic behavior model and the interaction protocol</td>
<td>Finished mapping</td>
</tr>
<tr>
<td>d4</td>
<td>Static behavior pre-/post-condition mismatch with event guards in dynamic behavior model</td>
<td>Starting blank model</td>
</tr>
</tbody>
</table>

To validate our results, we built separately a reliability model from the existing implementation of the DeSiController component. This code-level reliability model is based on a directed graph that represents the component’s control structure. We assumed that the implementation was faultless, so we injected defects, such as those shown in Table 1, into the code to simulate failure behavior. We built a Markov model by leveraging this graph, where a node in the graph translates to a state in the Markov model, analogously to what existing approaches have done at the system level (e.g., [2, 8]). We used the results obtained from this implementation-based model as the “ground truth” in our evaluations.

As described in Section 3, our framework allows for multiple failure classes. However, for clarity of exposition of results, in what follows, experiments are performed using one active class of defect at a time. In the presented experiments, this is done by setting probabilities of failures associated with the remaining defect classes to zero. That is, these experiments use only single failure state models, where the failure state corresponds to the class of defect being studied. We have also performed similar experiments where failure probabilities associated with defect classes other than the one under consideration are held constant at non-zero values – these correspond to multiple failure state models. The results of those experiments showed qualitatively similar trends to the results presented below and are available in [25].

### 4.1. Sensitivity to Information Sources

The first step in evaluating our framework was to perform sensitivity analysis on models built using different information sources. Our goal was to study how different information sources affect reliability prediction, rather than show which source is “best”. One set of experiments focused on a model’s sensitivity to component reliability when recovery probabilities change. To this end, we fixed the failure probabilities, and varied recovery probabilities from 0.1 to 1.0, at 0.1 intervals. We repeated this for different failure probabilities (from 0.05 to 0.2, at 0.05 intervals). The following information sources were considered in these experiments.

**Case (1) – Domain Expert** – We relied on the information provided by DeSi’s primary developer, and explored only the operational profiles suggested by him.

**Case (2) – Simulation** – We were provided with DeSi’s requirements [13], based on which we specified a sequence of high-level events to simulate the dynamic behavior model of DeSiController shown in Figure 4a. We obtained training data by leveraging the simulation trace and applied our HMM-based approach to obtain behavioral transition probabilities (recall Section 3.2).

![Architectural models of the DeSiController component at different levels of detail](image-url)
Case (3) – Functionally similar component – We obtained training data from an older version of DeSi that was missing certain functionality. We again applied our HMM-based approach to obtain behavioral transition probabilities.

Our results are presented in Figure 5, where we plot component reliability as a function of recovery probability corresponding to the defect class under consideration. Each curve in the figure corresponds to a different failure probability, \( p \), again, corresponding to the defect class under consideration. Specifically, we activate defect \( d_1 \) from Table 1 in Figure 5a, defect \( d_2 \) in Figure 5b, defect \( d_3 \) in Figure 5c, and defect \( d_4 \) in Figure 5d. We observe that the trends conform to our expectations in all four cases: as recovery probability increases, the reliability of the component increases; moreover, as failure probability increases, component reliability decreases. We also note that even when the recovery probability is 1, the reliability of the component is less than 1. This is because failures can still occur: failure probability is not zero and recovery from a failure is not instantaneous.

Although the general trends across the experiments are similar, Figure 5 yields some interesting observations. First, the sensitivity of the Case (1) results, and their accuracy as compared to the code-level model results, varies depending on the defect being studied. This and several other similar examples indicate that information provided by an expert may be inaccurate, or that in practice the component may not behave as expected. Relying on expert opinion alone in estimating architecture-level reliability, as most existing approaches appear to do, can therefore be error prone.

Another observation is that in Figure 5b, reliabilities in Case (3) are very high. This is because the older, functionally similar version of DeSi does not have the functionality that generates a deployment automatically based on user constraints. As a result, defect \( d_2 \) could never happen in this older version of DeSiController. Similarly, in Figure 5d, Case (3) exhibits different sensitivity than results obtained using other information sources. This is because users rely more on creating deployments manually in DeSi’s older version, hence defect \( d_4 \) occurs more often in the older version, ultimately resulting in lower reliability values. This illustrates the fact that a functionally similar component is only useful in predicting reliability for the functionality that is available and used in a comparable fashion in both components. Information from other sources will be required to predict the effect of newly added functionality on certain defect classes.

We also note that in the experiments of Figure 5, the code-level model exhibits higher reliability than the other cases. This occurs because the code-level model is finer-grained than the architectural models. As we will show in Section 4.3, coarser-grained models give more conservative results in our framework. We will also discuss why this is a desirable property of the framework.

In summary, the results shown above corroborate our assertion that in order to provide a meaningful evaluation of a component’s reliability, having information from multiple sources is desirable: information from certain sources may be unavailable (e.g., functionally similar component) or inaccurate (e.g., expert opinion). As part of our future work, we plan to further explore this hybrid approach. For example, in the context of DeSiController, we can potentially improve our results in Case (3) of Figure 5b by simultaneously using a functionally similar component and domain knowledge to estimate the operational profile that corresponds to the old and new functionalities, respectively.

4.2. Sensitivity to Operational Profile

To evaluate our reliability framework’s sensitivity to changes in a component’s operational profile, one approach we have taken is to fix the transition probabilities among all states of the component’s reliability model (recall Figure 2b), except for a specific set. By varying those remaining transition probabilities, we can observe the model’s response. In this section, we will consider the ranges of DeSiController’s reliability values when the probability of going from state Finished mapping to state Waiting for command (recall Figure 4a) varies between 0 and 1, while all other parameters in the operational profile are fixed. This corresponds to estimating the average number of iterations of DeSiController’s deployment calculation algorithm.

We reiterate that the same analysis was performed by varying transition probabilities between other states, and yielded qualitatively...
similar results. We varied the failure and recovery probabilities (as in Section 4.1), and obtained a reliability range for each failure-recovery probability pair, for all four defects from Table 1.

Figure 6 depicts our results. Each graph in this figure represents a case with a given failure (fp) and recovery (rp) probability. The horizontal bars represent the range of reliability values obtained by varying the probability of going from state Finished mapping to state Waiting for command between 0 to 1. The bars labeled (i), (ii), (iii), and (iv) represent the defects d1, d2, d3, and d4, respectively. We observe that the reliability ranges are larger when failure probabilities increase and/or recovery probabilities are lower. This corresponds to the graphs concentrated toward the left and bottom portions of Figure 6. This means that, when failures occur more frequently and/or are harder to recover from, the component’s reliability is more sensitive to the specifics of the operational profile.

Another observation is that DeSiController’s reliability was most sensitive to defects d1 and d2. This is because d1 and d2 directly affect the two states on which we focused in this particular scenario. More generally, by varying operational profiles, we can identify which defects most prominently affect the resulting reliability values across these operational profiles. If a defect is shown to increase the model’s sensitivity to multiple operational profiles, software architects may want to focus their attention particularly on eliminating that defect in order to achieve the greatest improvement in the component’s reliability.

4.3. Sensitivity to Model Granularity

Software architectural models may vary widely in terms of the amount of detail they contain. Different models are produced at different points during the system’s development, and may be intended for different stakeholders. On the average, it is possible to produce high-level models earlier than detailed ones during a system’s development; it is also easier to discover and mitigate any design flaws in them. On the other hand, a high-level model may not be representative of a system’s or component’s complexity and, as we will elaborate below, it may obscure defects that can easily creep in during design refinement and implementation. Our goal is to study the effects of model granularity on reliability prediction, rather than to help system modelers choose what model they should use; that decision will depend on the development context.

In our case, the objective is to assess the impact that the amount of detail in a component’s architecture-level model has on the component’s reliability calculated using our framework. To this end, we have performed sensitivity analyses on component models of varying granularity levels. For example, Figure 7 shows the results of calculating the reliability of the DeSiController component based on its models at the three levels of granularity from Figure 4, with injected defect d3 from Table 1 and its operational profile estimated by the DeSi expert. Again, we plot reliability as a function of recovery probability from d3-related failures, and the different curves correspond to failure probabilities due to d3. Performing this analysis using other information sources (functionally similar component and simulation) and other defects consistently yielded qualitatively similar results; we omit them due to lack of space.

The detailed model of DeSiController from Figure 4a is the one we have used in all of our measurements discussed in the preceding subsections. Two higher-level models of the same component, developed with the help of DeSi’s designers, are depicted in Figures 4b and 4c. Note that each state in a higher-level model relates to multiple states in a finer-grain model. For example, the Running greedy algorithm state in the model shown in Figure 4b abstracts away the portion of the state machine comprising the four states and their transitions in the upper right segment of Figure 4a.

We observe that, when recovery probability is fixed while failure probability increases from 0.05 to 0.2, reliability values are most sensitive in the highest-level model (corresponding to Figure 4c). Another observation is that the model from Figure 4c is more sensitive to recovery probability than the model from Figure 4b, while the most detailed model (Figure 4a) is least sensitive. This can be explained as follows. Failures corresponding to defect d3 only take place in the Finished mapping state of the 24-state model (Figure 4a). On the other hand, time spent in the Running algorithm state in the 11-state model (Figure 4b) also includes the time that, in the 24-state model, would be spent in the Ranking unmapped hosts and Ranking unmapped components states. As a result, the sensitivity is higher in the 11-state model. An analogous argument holds for the difference in sensitivity between the 5-state and 11-state models.

The above also suggests that it is easier to narrow down the exact sources of defects using a detailed model. For example, defects associated with the middleware adaptor in DeSiController (the Processing middleware command state in the 11-state model of Figure 4b) may have been overlooked in the 5-state model. This is because the processing of all user-level commands in the 5-state model is described in a single state – Processing command. A high-level model of a component can still be very useful in that it can provide a conservative and quick prediction of a component’s reliability because smaller models require less computation.
Note that in our experiments a model with fewer states gives more pessimistic results. We argue that, in general, it is (a) desirable to provide more conservative predictions given less information and (b) necessary to do so consistently. This will, both, sensitize engineers to the potential problems the system may eventually exhibit, and provide confidence in the framework’s predictive power.

5. CONCLUSIONS

Meaningful architecture-level reliability prediction is critical to the cost-effective development of complex software systems. However, early efforts in this area have assumed some degree of knowledge of individual components’ reliabilities and operational profiles. In this paper, we have argued that these assumptions are not reasonable. We have presented a framework for component-level reliability prediction that does not rely on such assumptions.

We approached the challenges associated with the lack of information about a system and its components early in development by (a) exploiting behavioral models that already exist as part of the software architecture design process, (b) exploring the sources of information available at design time, and (c) coupling these with stochastic modeling techniques that have been successfully applied in dependability modeling. The complexity of each phase of the resulting approach (recall Section 3) is as follows. Phase 1’s complexity is a function of the chosen architectural analysis technique, which is independent of the reliability prediction approach used. Phase 2’s complexity is a function of the approach used to estimate transitions (e.g., use of domain expertise is not as computationally costly as the use of HMMs, but in general this process is polynomial in the size of the state space). Finally, Phase 3’s complexity is a function of the approach used to solve a system of linear equations (typically, cubic in the size of the state space).

Our evaluation results indicate that our framework provides meaningful reliability prediction in the context of early stages of software development. Our on-going research is focusing on exploring hybrid approaches to unifying information from different sources. At the same time, we believe that scalability of reliability prediction techniques at the system level remains a challenge, and we are also targeting our work at addressing this problem.

6. ACKNOWLEDGEMENTS

This work is supported by the NSF (award numbers 0312780, 0417274, 0509539, and 0720612). The authors wish to thank the anonymous reviewers for their valuable comments.

7. REFERENCES