

CS377 – Fall 2009

Homework Assignment 6

Due Date: December 1, 2009 before 12:30pm

As an engineer, you will be forced to update your skills continuously, by reading technical literature, attending conferences, watching presentations, and, of course, coming up with your own ideas. There is a huge, and quickly growing, body of literature on the myriad aspects of software engineering in particular. There are so many papers published, new magazines and journals established, and workshops, symposia, and conferences organized and held that you could probably spend your entire professional life just reading and listening to other people's ideas about the field, yet still would not have the time to get exposed to anywhere near all of them. Even if you were able to ingest them all, a great majority will prove over time to be bad ideas, or will quickly be supplanted by better solutions that make them obsolete. Yet, without exception, you will find yourselves sooner or later being "sold" some new approach by a colleague or your manager as "the next big thing". Deciding relatively quickly, based on objective criteria, whether that is really the case and whether, and to what extent, you need to alter the way you are conducting your professional life will be a highly valuable skill that may save you a great deal of time and effort.

In this assignment, you will be given two versions of the same paper, by the same authors, on the same topic. The authors are elided to ensure a "double-blind" review, where neither the authors know who the reviewers are, nor do the reviewers know who the authors are, in order to prevent any bias. The topic should be sufficiently familiar to you to complete this assignment even though you are not expected to understand all the nuances of the technical approach (e.g., the equations, etc.). Imagine that someone just dropped one of these papers on your desk with a post-it note that said "I think this would be a good idea for our project". How would you know what to answer them?

This is all the more difficult because the two papers are quite similar, yet the initial version of this work was rejected from a top scientific conference a couple of years ago, while the modified version was not only accepted, but was in consideration for the *Distinguished Paper* award at another top conference later that same year. Your assignment is to critique the two papers, first as stand-alone pieces of work and then in comparison to one another. This means that you will produce three separate critiques.

I am interested in your views, not in the papers' original reviewers' views. This means that I do not want you to try and find out which of these versions was actually published. If you try to do that, you will be doing extra work unnecessarily, and in the end may fail to identify the "winner" anyway since the two papers are quite similar. The objective of the exercise is to help you appreciate how the difference between success and failure in a technical field such as software engineering is often a very fine line, and may not even have to do with technical factors. So, we will refer to these papers simply as *Paper 1* and *Paper 2*.

The individual critiques should contain (1) the paper's grade—A, B, C, or D, (2) a summary of the main idea (usually one paragraph); and (3) an evaluation of the paper, with the key strengths and weaknesses clearly indicated. The evaluation need not be too long, but it should be thorough. In other words, you do not need to (and, in fact, should not) repeat the points already made in the paper. Instead, I am interested in your view of the paper, which is a set of things I may not have

necessarily gotten out of it by reading it myself. As an example to help guide you, I will give you two sample reviews of other papers—one generally positive and the other generally negative—that follow the same structure.

The third, comparison critique should only state which of the two papers you find stronger, in what respects, and why. This can be as short as a paragraph.

You will be graded on your ability to think critically about a presented idea, even when you do not have all of the necessary technical background to fully evaluate it. You will be expected to refer to the concepts discussed in class in formulating your reviews. You may comment on the two papers' respective clarity and quality of presentation, but that should not be the predominant reason why you choose one over the other – keep in mind that the same authors wrote both papers, and presumably did so equally carefully.

Sample Review #1

Grade: B

Summary: The author considers the source code for four different OS kernels: FreeBSD, Linux, OpenSolaris, and Windows Research Kernel (WRK). He performs different source code-level analyses and gathers a large number (close to 50) of different metrics for each of the kernels. He then justifies and interprets each of the metrics after having collected them from the four OSs. Finally, he draws some general conclusions from the study, which I will restate and summarize:

1. Engineering requirements for a non-trivial software system trump the development process used.
2. Several characteristics of the source code (e.g., reliance on preprocessing, code structure vs. style disparities) can be tied to the development context (e.g., open source enthusiasts vs. a more rigid commercial environment).
3. Open source projects do not produce markedly higher quality code than proprietary development projects.

Evaluation: The paper is very well written. The problem is motivated properly, the metrics described, applied, and then interpreted individually quite well. The subject matter is relevant. The study is clearly sizable and its conclusions potentially useful and influential. At the same time, I found the paper's overall conclusions (summarized above) either tangential to what I would consider the key issues (#1), unsurprising (#2), or simply unsupported by the study (#3). The latter issue is the most serious one in my view. What is this conclusion based on? One huge facet of software quality is its runtime behavior: availability, reliability, robustness, etc. I am not sure how any claims about the quality of a system can be made from the code alone.

Also, what immediately jumped out at me is that the size of WRK, as measured in SLOC, is smaller than the other OSs by factors of between 3 and 5; its number of macros by factors of up to 20; its number of C functions by up to an order of magnitude, etc. Why was this not considered to be important?

Additionally, the author generalizes his results in the paper's last section, and in a couple of cases I found his generalizations tenuous. For example, providing Table 2 and then asserting that it somehow demonstrates that the four OSs were ultimately comparable in terms of the chosen metrics begs many questions, the most important of which, to me, was: what is that assertion based on? The relative numbers and distribution of + s - s and blank spaces could just as easily have been interpreted (very) differently.

In the end, I liked the paper, its aims, and even its methodology (although, as I said, I am not convinced that the quality of a software system can be ascertained by looking at its source code), but found its conclusions either underwhelming or off .

Sample Review #2

Grade: D

Summary: This paper argues that, in order to adequately understand the defect-related behaviors of software systems over their lifetimes, such as the systems' reliabilities, defect repair times must be properly modeled. Previous approaches use either a lognormal (LN) or an exponential (EXP) distribution to characterize defect repair times. The authors hypothesize that a different distribution, which they call the Laplace Transform of the lognormal (LTLN), is more appropriate as the mathematical model of defect repair times. They try to demonstrate their claim by studying defect data from several product lines at Cisco.

Evaluation: This was not an easy paper to get through. Part of the difficulty certainly has to do with the subject matter. However, a part is also due to the paper's organization, writing style, lack of a bigger picture, lack of appropriate motivation, background information, implications of the choices made by the authors, and so on. From the start, the authors seem to assume certain things that a reader (this reader for example) may not agree with, or at least may not find nearly as obvious. For example, the entire motivation for the paper appears to rest on the claim that defect repair time is a more tangible metric, whose impact on reliability and development schedule can be understood directly than defect repair rate. There is absolutely no justification for this claim however. If defect repair rate can be obtained and modeled adequately using an LN distribution, how novel is the observation that all that's needed then to transform it to a defect time distribution is to apply the Laplace transform -- that is already a known fact.

Throughout the paper I felt as if the authors are not saying more than they are actually saying. For example, what is the relevance of the applicability of this work to defect subsets? Why all the references to the work on the lognormal distribution -- it wasn't in any way tied to the rest of this paper. For example, the entire Section 2.1 could have been summarized in one sentence (the first sentence of the last paragraph).

Similarly with the discussion of multiplicative factors (Section 2.2): a great amount of detail is given for what ultimately turned out to be only an illustrative set of possible such factors. The tie-in to COCOMO seemed strange and off topic, while the claim that this section shows how a multiplicative interaction of these factors leads to a lognormal distribution of repair rates was dubious. The values given in Table 1 seemed arbitrary and I would have expected something more than the claim that they come from prior research (which prior research?) and anecdotal evidence (which evidence?). Furthermore, the mathematical formulation was unclear/unexplained. Several parameters in equation 1 were discussed extensively, but clarified/explained in later sections.

The introduction of terminology at random times also didn't seem to help. For example, the revelation that previous research demonstrated that LN is better than Weibull and gamma distributions for modeling defect repair times somehow to me does not belong in a section titled Limited conditions of the LTLN.

The application of the AIC in Tables 3, 4, and 5 was ambiguous at best. What does it mean that Two units of AIC is significant, four very significant. ? Also what is the exact meaning of the column AIC LTLN vs LN. Does it show their difference or something else? And how should the difference be interpreted? This information is completely missing, yet the authors go to great lengths to take apart the data as pertaining to equations 1 and 2. The fact that the quality of writing degraded quite a bit in this section (e.g., provide orient the reader, are generally are more virulent etc.) did not help.

The bottom line, and the biggest issue the paper would need to address is: why does one care? What is the impact of this work, e.g., as applied to an existing reliability model? How significant is the AIC difference between LTLN and LN? How significant is the fact that LTLN doesn't always outperform LN? What can a software engineer get out of this work? In addition to significantly improving the presentation, motivation, and clarifying the evaluation, in my view it is imperative that these questions be answered before the paper can be published.

PAPER 1 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.)

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 of one of 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 the 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 state-chart 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, 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 opera-

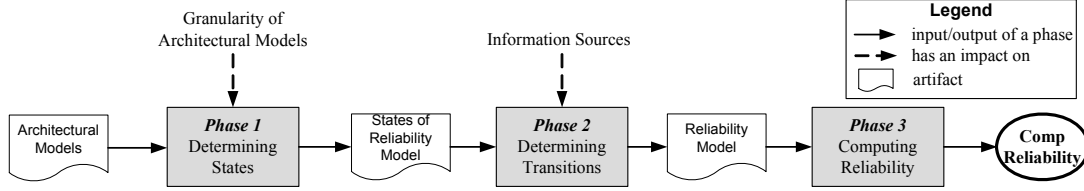


Figure 1. Software component reliability prediction framework

tions; the *dynamic behavior* shows a continuous view of the component’s internal execution details; and the *interaction protocol* provides a continuous external view of a component’s execution by specifying the allowed execution sequences of its operations (accessed via interfaces). We used 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 $S = \{S_1, S_2, \dots, S_N\}$ 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.¹ 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.

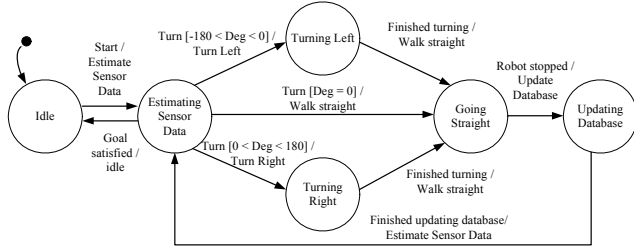
Modeling Assumptions. In addition to modeling normal and failure behavior of a component, as in existing approaches (e.g., [2]), we are also interested in modeling recovery behavior. Hence, we can define reliability as the probability that the component is in a state representing normal behavior. Moreover, 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 SCROver, 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. SCROver 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.

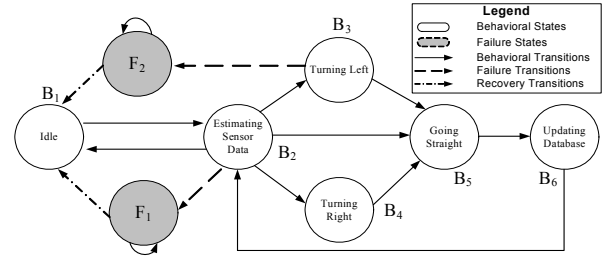
3.1. Phase 1: Determining States

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.

1. Note that, while our focus is on the component level, determining failure modes and transition probabilities is also difficult at the system level, and this problem is not properly addressed in existing literature.



(a) Dynamic behavior model



(b) Reliability model

Figure 2. The Controller

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* (B_1), *Estimating Sensor Data* (B_2), *Turning Left* (B_3), *Turning Right* (B_4), *Going Straight* (B_5), and *Updating Database* (B_6). 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.

For example, we identified two defects in one version of the *Controller* component, shown in Figure 3. Defect d_1 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 a *Go Straight* event while it is in the *Estimating Sensor Data* state, which in turn may adversely affect the component’s reliability. Defect d_2 is shown on the right side of Figure 3: a negative value of deg represents turning left in the dynamic behavior model, while in the static behavior model, deg can never be negative. When the *Controller* component interacts with an actuator, this

defect may result in unexpected behaviors when the robot passes a negative value of deg as a parameter.

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 d_1 than why the robot did not turn appropriately as a result of defect d_2 .

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 F_i . 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 d_1 and d_2 identified in the *Controller* component are assigned to two different classes D_1 and D_2 , respectively. Therefore, as depicted in Figure 2b, we designate two failure states $F = \{F_1, F_2\}$ 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

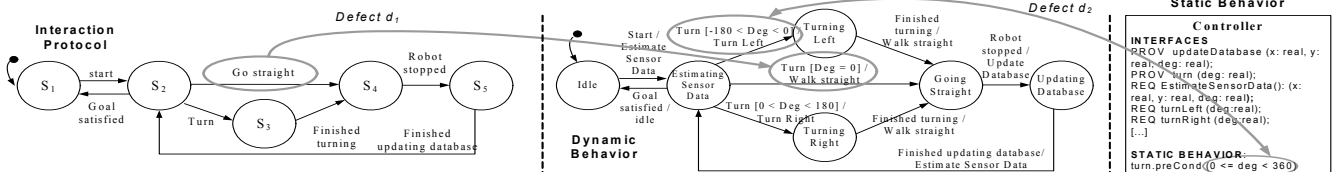


Figure 3. Examples of architectural inconsistencies

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 *information sources* that may be available at the architectural level.

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 $S = \{S_1, S_2, \dots, S_N\}$, a transition matrix $A = \{a_{ij}\}$ representing the probabilities of transitions between states, a set of observations $O = \{O_1, O_2, \dots, O_M\}$, 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_l after recovery from a defect of class k .² 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_j , 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 probabilities 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 component'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 $i = F_1, \dots, F_M, B_1, \dots, B_N$. As t goes to infinity (i.e., as the component operates for a long time), these probabilities converge to a stationary distribution,

$$\hat{\pi} = [\pi(F_1), \dots, \pi(F_M), \pi(B_1), \dots, \pi(B_N)]$$

which is uniquely determined by the following equations:³

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

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

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:

$$R = 1 - \sum_{i=1}^M \pi(F_i) \quad (2)$$

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-

$$P = \begin{matrix} & F_1 & F_2 & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ \begin{matrix} F_1 \\ F_2 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \end{matrix} & \begin{bmatrix} 0.8 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0.05 & 0 & 0.019 & 0 & 0.076 & 0.0095 & 0.8455 & 0 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0 & 0 & 0.96 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix} \quad (3)$$

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 effectiveness 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 reliability 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 mitigation 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 system [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 collaboration 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 predictions 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 evaluations 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-

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

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 1: Defects injected in *DeSiController*

Defect	Description	Affected State
d_1	Mismatched signatures	Waiting for command
d_2	Missing model validation rules in the design document	Validating model
d_3	Mismatch between the dynamic behavior model and the interaction protocol	Finished mapping
d_4	Static behavior pre-/post-condition mismatch with event guards in dynamic behavior model	Starting blank model

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).

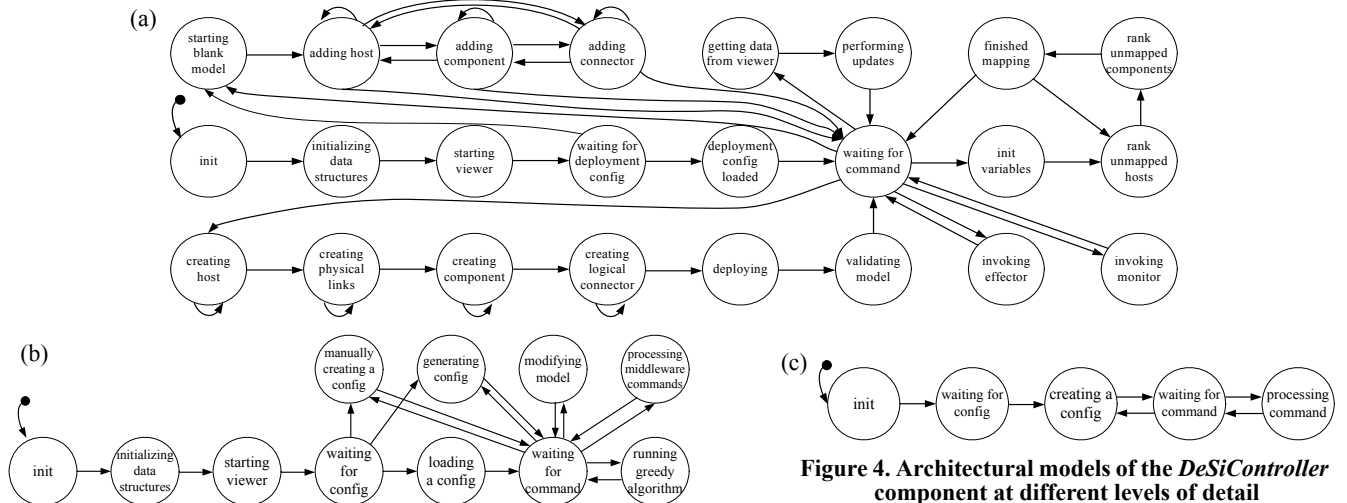


Figure 4. Architectural models of the *DeSiController* component at different levels of detail

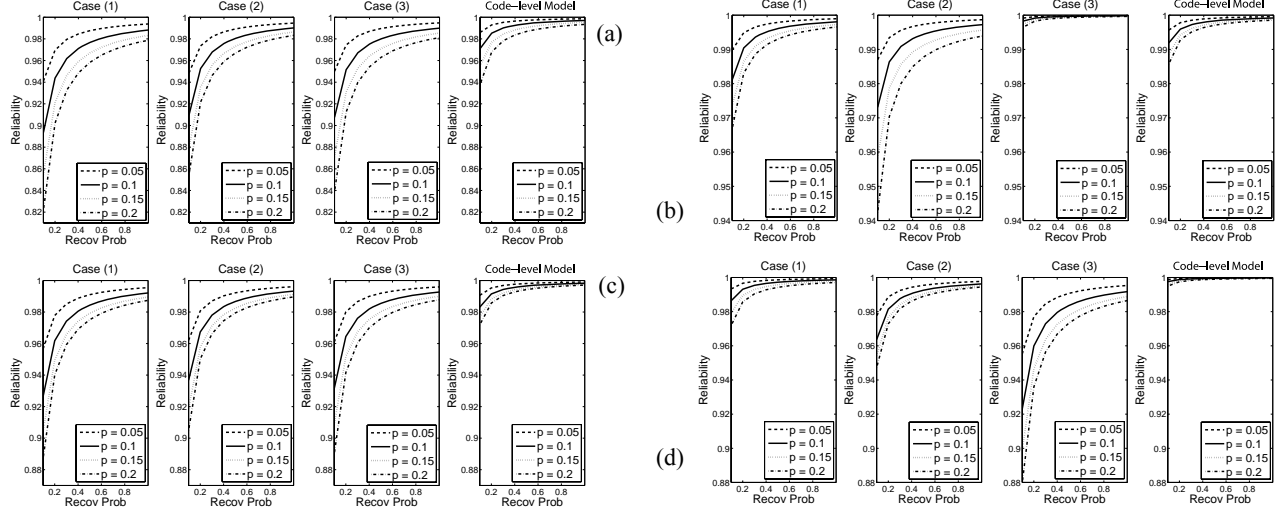


Figure 5. Analysis of sensitivity to information sources

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 for all defects: 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

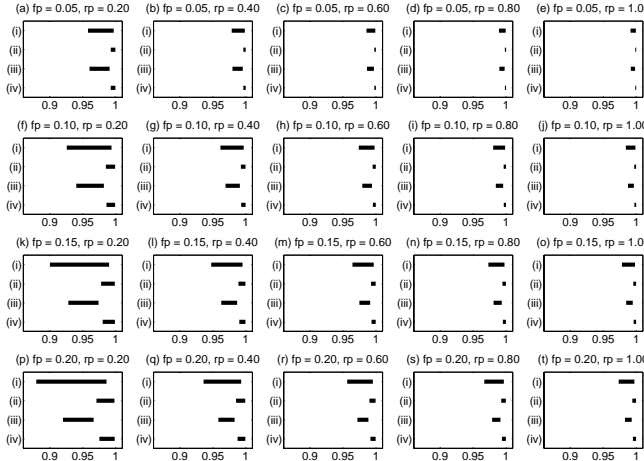


Figure 6. Analysis of sensitivity to operational profiles

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 d_1 , d_2 , d_3 , and d_4 , 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 d_1 and d_3 . This is because d_1 and d_3 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

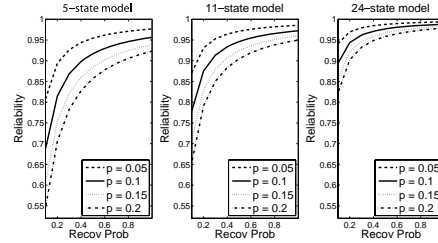


Figure 7. Analysis of sensitivity to models of different granularities

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 d_3 from Table 1 and its operational profile estimated by the DeSi expert. Again, we plot reliability as a function of recovery probability from d_3 -related failures, and the different curves correspond to failure probabilities due to d_3 . 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 d_3 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 model.

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.

7. REFERENCES

- [1] Boehm B., et al. Using Empirical Testbeds to Accelerate Technology Maturity and Transition: The SCRover Experience, in *Proceedings of ISESE'04*, pp. 117 - 126, 2004.
- [2] Chung, R.C. A User-Oriented Software Reliability Model, *IEEE Transactions on Software Engineering*, 6 (2), 1980.
- [3] Cortellessa V. and Grassi V. A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems. In *Proc. CBSE-10*, Jul 2007.
- [4] DeLine, R. Avoiding Packaging Mismatch with Flexible Packaging. In *Proceedings of the 21st Int'l Conference on Software Engineering*, Los Angeles, CA, May 1999.
- [5] Gokhale S., Architecture-Based Software Reliability Analysis: Overview and Limitations. *IEEE Trans. on Dependable and Secure Computing*: 4(1), Jan 2007.
- [6] Gokhale S., et al. Reliability Prediction and Sensitivity Analysis Based on Software Architecture. *Proc. ISSRE'02*, 2002.
- [7] Gokhale S., et al. Reliability Simulation of Component Based Software Systems, *Proc. ISSRE'98*, pp. 192-201, 1998.
- [8] Goseva-Popstojanova K. et al. Architectural Level Risk Analysis using UML, *IEEE Transactions on Software Engineering*, Vol.29, No.10, October 2003.
- [9] Goseva-Popstojanova K., and Kamavaram S. Software Reliability Estimation under Uncertainty: Generalization of the Method of Moments, in *the Proceedings of the 8th IEEE Intl. Symposium on High Assurance Systems Engineering*, 2004.
- [10] Goseva-Popstojanova K. et al., Architecture-Based Approaches to Software Reliability Prediction, *Int'l J. Computer & Mathematics with Applications*, 46(7), October 2003.
- [11] Immonen A., and Niemela E. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, Jan 2007.
- [12] Malek S., et al., Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In *Proc. ICSE 2007*, May 2007.
- [13] Malek S., et al. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In *Architecting Dependable Systems III*, LNCS, October 2005.
- [14] Medvidovic N., and Taylor R., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering*, 26(1), Jan. 2000.
- [15] Rabiner L.R., A Tutorial on Hidden Markov Models, in *Proceedings of the IEEE*, vol. 77, pp. 257-286, 1989
- [16] Reussner R., et al. Reliability prediction for component-based software architectures, *J. Systems and Software*, 66(3), 2003.
- [17] Roshandel R., et al. Understanding Tradeoffs among Different Architectural Modeling Approaches. In *Proc. 4th Working IEEE/IFIP Conf. on Software Architecture*, June 2004.
- [18] Roshandel R., et al. Estimating Software Component Reliability by Leveraging Architectural Models. *Emerging Results track, ICSE 2006*, pp. 853-856, Shanghai, China, May 2006.
- [19] Roshandel R., Medvidovic N. Multi-View Software Component Modeling for Dependability, in *Architecting Dependable Systems II*, LNCS, 2004.
- [20] Roshandel R. et al. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. In *Proceedings of 3rd QoS*, Boston, MA, July 2007.
- [21] Stewart W.J., Introduction to the numerical solution of Markov Chains. Princeton University Press, 1994.
- [22] Wang W., Wu Y., Chen M., An architecture-based software reliability model, in *Proc. of Pacific Rim International Symposium on Dependable Computing*, 1999.
- [23] Yacoub S.M., et al. Scenario-Based Reliability Analysis of Component-Based Software, in *10th Int'l Symposium on Software Reliability Engr.*, Boca Raton, Nov. 1999.
- [24] http://sunset.usc.edu/~nen/cs477_2003/, CSCI477 – Design and Construction of Large Software Systems, USC.
- [25] <http://vista.usc.edu/~lccheung/reliability/>

PAPER 2 Abstract

The ability to predict the reliability of a software system early in its development, such as at the level of software architecture, can provide a cost-effective way of improving the system's quality. Predicting the reliability of individual components is an important first step in doing so for an entire system. Existing approaches typically assume the reliability of system components to be known. That assumption is unreasonable in general. Predicting a component's reliability at the architectural design level is challenging because of many uncertainties associated with the system under development. This paper presents a software component-level reliability prediction framework. The objective of the framework is to allow software architects to explore early on the effects of different design choices, and particularly of the introduced architecture-level system defects, on a component's reliability. The framework leverages a component's architectural model to construct and solve a stochastic reliability model. The framework is tailorable with different types of information that may be available during architectural design. It has been extensively evaluated for sensitivity to the uncertainties inherent in early development stages.

1. INTRODUCTION

Software reliability techniques are aimed at reducing or eliminating failures in software systems. Conventional software engineering wisdom suggests that assessing reliability (or any 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. If significant problems are identified during implementation or operation, large parts of the system may have to be reengineered, which has been shown to be prohibitively costly. Therefore, trying to *predict* a system's reliability early in its development is desirable and can potentially provide a more cost-effective way of improving a software system's quality.

It is widely accepted that *software architecture* is a linchpin in the development of complex software systems [14, 22]. Architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system. Architectural decisions directly affect aspects of system dependability, such as reliability. Identifying and mitigating design flaws early in development helps to increase dependability of a system in a cost-effective

manner. In order to achieve this goal, reliability and other quality attributes must be "built into" the software system throughout development, including during architectural design. This 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 [2, 7, 9, 24]. While they acknowledge that individual components' reliabilities have a significant impact on system reliability, these approaches invariably assume that (1) the reliabilities of the individual components in a system – in the case of modeling the system's reliability, or (2) the reliabilities of a given component's elements, such as its services – in the case of modeling a component's reliability, are known. We do not believe these assumptions to be reasonable. It is unclear how the reliability of a component, or of one of 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 have been implemented and one of the existing code-level reliability estimation techniques applied on it. None of these options is satisfying: the first for (hopefully) obvious reasons; the second because such an "oracle" may not exist or may itself be unreliable; the last because it directly precludes one from applying a reliability prediction technique during architectural design.

This paper strives to remedy the shortcomings of previous approaches. We propose a software component reliability prediction framework that leverages architectural models to construct and solve stochastic reliability models. We argue that architecture-level reliability prediction is a multi-faceted problem that is directly impacted by the many uncertainties present early in development, and by the lack of the necessary information about a system and its components. Specifically a component's *operational profile* and its *failure conditions* cannot be reliably obtained at the architectural level.

The lack of this information forces us to devise a way of deriving, combining, and applying other existing information sources available during architectural design. For example, (1) system engineers' and domain experts' intuitions and experience can be combined with (2) simulations of a component's behavior constructed from the architectural model, and (3) analysis of the execution logs of functionally similar components (e.g., from a similar system or a previous version of the system under construction). By leveraging such sources of information, we can produce candidate operational profiles for reliability prediction. To handle uncertainties associated with the lack of failure information, we perform

defect classification by analyzing inconsistencies within a component’s architectural model [17].

We have also studied how reliability prediction is affected when component models of *different granularities* are used. A software architect may choose to elaborate or elide certain details about the system being modeled depending on the current development context (e.g., whether the model is to be formally analyzed for correctness, passed on to developers for implementation, or discussed with non-technical stakeholders such as managers or customers).

Our framework is intended for design-time exploration, by allowing software architects to compare the effects of different design decisions, and especially of system defects caused by those design decisions, on reliability. We have applied the framework on a large number of software components. Specifically, we have evaluated the sensitivity of our reliability models to the sources and quality of the available information, as well as to the level of detail provided in the components’ architectural models. For validation purposes, we have also explored our framework’s ability to predict the reliability of already implemented components.

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 prediction—has in fact been borne out in practice. At the same time, our results indicate that it is possible to meaningfully assess a component (and, by extension, a system) for reliability even when the information is distributed, sparse, and itself not entirely reliable. Our framework has shown a high degree of predictive power and resiliency to changes in these parameters.

The rest of the paper is organized as follows. Section 2 reviews existing research that corroborates and/or 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. RELATED WORK

Software reliability modeling techniques have been broadly classified into white-box and black-box models [11]. The black-box models, such as software reliability growth models (SRGMs) [3, 8, 10], use statistical approaches to predict reliability of an implemented system and do not consider the system’s internal structure. Such techniques are not suitable for predicting reliability at the architectural level since they require information from system testing and execution. The white-box models, on the other hand, take the internal structure of a software system into account. A number of white-box approaches assume that system components have already been implemented, and hence one of the implementation-level technique can be applied to estimate component reliability. For instance, Shooman’s model [23] estimates system reliability based on the total number of failures obtained in a number of test runs. Gokhale et al. [4] predict reliability of a component based on data obtained from testing the system using a regression test suite.

Several white-box approaches are applicable at the architectural level; a survey of such approaches is presented in [7]. These approaches leverage architectural configurations of software systems and build reliability models by considering interactions between components. For example, Reussner et al. [16] build architectural reliability models based on both structural and behav-

ioral specifications of a system. Another example is Wang et al. [24], which focuses on reliability modeling of architectural configurations with heterogeneous architectural styles. However, with the exception of [5], even though the overall software system is modeled as a white box in these approaches, the system’s components are modeled as black boxes, and their reliabilities are assumed to be known [2, 6, 16, 24]. Moreover, these works (perhaps implicitly) assume that the operational profiles of the system are known. As is in the case of individual components, it is difficult to obtain the overall system’s operational profile at the architecture level. This is recognized in [6], which presents a study of uncertainty associated with unknown system operational profiles.

In a risk analysis technique proposed by Goseva-Popstojanova et al. [5], a component’s reliability risk is defined as a function of the component’s complexity and severity levels of its failures. The failure severity levels are assumed to be known. A component’s complexity is obtained by counting the number of nodes and transitions in its control flow graph. While it is true that the internal structure of a component contributes to its reliability risk (at least to some extent), a component’s dynamic behavior has a significant effect on its reliability as well. Since a component’s dynamic behavior is largely ignored in [5], we consider theirs to be a “grey-box” approach at the component level.

To the best of our knowledge, no other white-box component reliability estimation approaches have been proposed in the literature, aside from our initial efforts on this topic. We hypothesize that by treating components as white boxes, we can mitigate a number of uncertainties associated with architecture-level reliability modeling, which would in turn lead to better and more useful component-level (and eventually system-level) reliability estimation. The remainder of this paper describes our research and evaluation efforts in support of this hypothesis.

3. COMPONENT RELIABILITY PREDICTION 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* view of a component shows its provided and required services; the *static behavior* shows the functionality of the component 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* shows a continuous view of the component’s internal execution details; and the *interaction protocol* shows a continuous external view of a component’s execution by specifying the allowed execution logs of its operations (accessed via interfaces). These models are used as the starting point for our component reliability prediction framework.

For ease of exposition, we present our framework as a three-phase process depicted in Figure 1. Broadly, our framework leverages architectural models of a component to construct a stochastic process model of 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, firmware, 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,

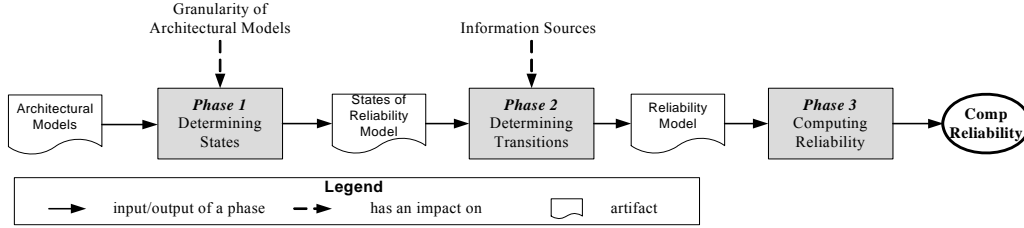


Figure 1. Component reliability prediction framework

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 [7, 16, 24].

Briefly, a discrete time Markov chain is a stochastic process with a set of states $S = \{S_1, S_2, \dots, S_N\}$ 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).

Two types of states in the set S need to be determined, corresponding to normal operation and to faulty behavior. States corresponding to normal operation can be obtained without difficulty from already existing architectural models, e.g., by considering the dynamic behavior model of a component. However, states corresponding to faulty behavior are typically not explicitly represented in architectural models, as components are not designed to fail. This is the more difficult part of the state determination process. In this work, we address this problem through the use of a defect classification technique [17] which looks for inconsistencies in a component’s architectural models. This is a tailorable element of our approach, i.e., other defect identification techniques can be substituted. The details of this state determination process (Phase 1 in Figure 1) are described in Section 3.1.

Given the states, what remains is the determination of transition probabilities between these states (i.e., the matrix P). A critical difficulty here is the lack of information (at the architectural level) about the operational profile of the component. In this work, 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 somewhat atypical use of Hidden Markov Models (HMMs) [15]. The description of information sources typically available at the architecture level as well as the details of transition probabilities determination (Phase 2 in Figure 1) are described in Section 3.2.

Once the states and the transition probabilities of the Markov chain reliability model are determined, to compute a reliability prediction, one must solve this model. 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 (Phase 3 in Figure 1) as described in Section 3.3. A number of approaches can be taken to insure tractability if the state space size is determined to be too big (either due to space or computational constraints) for some applications [21]. However, these are outside the scope of this paper.

Running Example. The example that we will use in this section is that of the *Controller* component of SCROver, 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. SCROver 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

For ease of exposition, we view the states in the set S as being of two types: (1) behavioral B , i.e., those related to the intended functionality of the component (obtained from architectural models), and (2) failure F , i.e., those related to the improper behavior of the component (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 state machine diagrams). The dynamic behavior model of the *Controller* component is illustrated in Figure 2a and consists of five states: *Initializing* (B_1), *Estimating Sensor Data* (B_2), *Going Straight* (B_3), *Turning* (B_4), and *Updating* (B_5). In our example, we map the states of the *Controller*’s dynamic behavior model to the behavioral states of the Markov chain reliability model (Figure 2b).

To determine the failure states (set F) we perform a defect classification by analyzing the architectural models of a component. In this paper, we use the defect classification technique described in [17]. As already noted, our framework can accommodate other such techniques. The multi-view approach to modeling a component described in [19] allows for the detection of architectural inconsistencies, which can be leveraged to represent defects. Defects identified in this stage contribute to the unreliability of the component. For example, we identified three defects in the *Controller* component: an interface mismatch with the sensor (d_1), parameters passed in the wrong order to the underlying firmware, which caused the robot to turn in the wrong direction (d_2), and an interaction protocol mismatch with the database (d_3).

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.

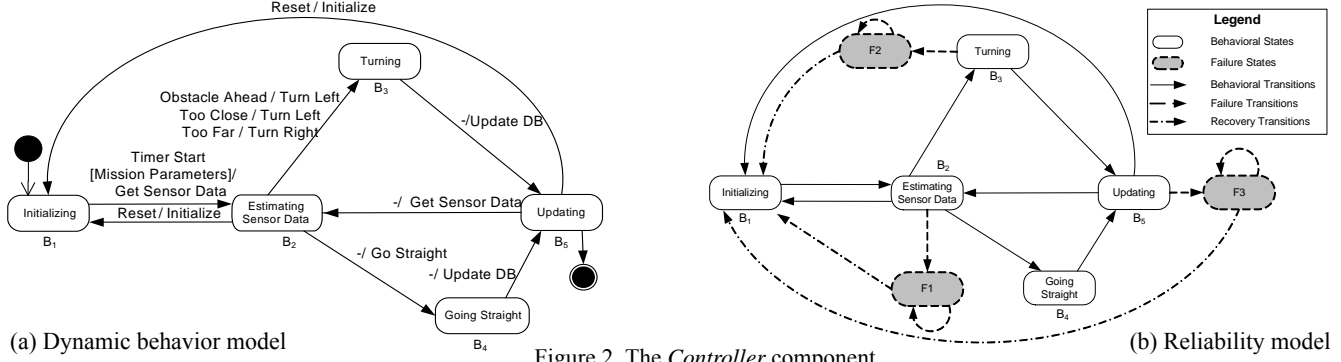


Figure 2. The *Controller* component

For example, it was much easier to uncover why the robot was unable to avoid obstacles (d_1) than why the database occasionally contained stale data (d_3).

In our reliability model, we represent the i -th defect class by a failure state, F_i . Making decisions about 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 another extreme, we could assign each defect its own defect class; this would result in one failure state per identified defect. One could also do something “in-between”, e.g., by grouping similar types of defects into a single defect class. 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. As illustrated in Section 4, this flexibility allows exploration of the effect(s) an individual defect (or a group of defects) has on the component’s reliability. In the *Controller* example, we use a failure state for each identified defect as depicted in Figure 2b, i.e., $F = \{F_1, F_2, F_3\}$, corresponding to defects d_1, d_2 , and d_3 , respectively.

3.2. Phase 2: Determining Transitions

The transitions in our Markov chain model (i.e., corresponding to the elements of matrix P) can be viewed as being of three different types: (1) behavioral, i.e., between two behavioral states, (2) failure, i.e., from a behavioral state to a failure state, and (3) recovery, i.e., from a failure state to a behavioral state. In what follows, we refer to transition probabilities as being behavioral transition probabilities, failure probabilities, and recovery probabilities. The process of determining these probabilities may differ, depending on the information available to a modeler. The following information sources may be available at the architectural level.

Domain Knowledge. We can obtain information about a component from a domain expert. One disadvantage, as discussed before, is that such an expert may not be available. Even if an expert is available, she may provide inaccurate information, either due to the complexity of the component’s description or due to unexpected operation 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 [4] 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, such as a domain expert or the requirements.

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.

We note that several of the above information sources may be available simultaneously. In that case, we can use them in a complementary manner so as to mitigate their different disadvantages. For example, if a domain expert is available, we can ask her to provide operational profile-related information. This would provide the input needed for the architectural simulation, which would, in turn, eliminate the expert’s inaccuracies.

Determining Behavioral Transition Probabilities. Let us define q_{ij} to be the probability of going from state B_i to state B_j , where both B_i and B_j are behavioral type states. 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. This corresponds to determining q_{23} , the probability that the robot turns.

If domain knowledge is available, instead of considering the entire set of possible operational profiles, we can focus on the subset 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 focus on a subset of operational profiles corresponding to relatively small values of q_{23} .

When traces are available, either from simulating a component’s architectural models, or from a functionally similar component, we

can use them to obtain the behavioral transition probabilities. A Markov model-based approach would typically assume that there is a one-to-one correspondence between observed events in the logs and transitions in the model. However, such correspondence may not exist, especially in the case of a functionally similar component. For example, in the *Controller* component, when we observe the *Reset* event, we cannot tell whether a transition occurred from the *Estimating Sensor Data* state or the *Updating* state. Here, we hypothesize that in such a case we can use Hidden Markov models (HMMs) [15] as one approach to estimating behavioral transition probabilities. HMMs assume that, while the number of states in the state-based model is known, the exact sequence of states needed to obtain a sequence of observed events may be unknown. The challenge is to determine the hidden parameters from the observable parameters.

An HMM is defined by a set of states $S = \{S_1, S_2, \dots, S_N\}$, a transition matrix $A = \{a_{ij}\}$ representing the probabilities of transitions between states, a set of observations $O = \{O_1, O_2, \dots, O_M\}$, and an observation probability matrix $E = \{e_{ik}\}$, which represents the probability of observing event k given that we are in state 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]. This initialization can be done more intelligently (by utilizing architectural models), which in our experience results in faster convergence of the HMM training; due to lack of space, we omit these details here.

An HMM can be trained to “learn” the unknown parameters in the model. The commonly used method to train an HMM is the application of the Baum-Welch algorithm [15]. An input for this training process is called training data, and typically consists of sequences of observations. In the context of our framework, we can use simulation traces (when simulation of architectural models is available) or execution traces (when a functionally similar component is available) to generate training data. Given an initial HMM constructed as described above, the Baum-Welch algorithm constructs an HMM that has a high probability of generating the input 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., $a_{ij}=q_{ij}$ for all i and j .

We note here that the training data artifact is assumed to be defect-free, i.e., the observations in the training data do not include any failure and recovery behavior. This assumption 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 , i.e., in the reliability model, f_{ij} is the probability of going from a behavioral state B_i to a failure state F_j . We define r_{kl} to be the probability that the component enters state B_l after recovery from a defect of class k , i.e., in the reliability model, r_{kl} is the probability of going from a failure state F_k to a behavioral state B_l .¹ 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 defect classification process, as described in Phase 1. Also, for each defect class 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, i.e., for each behavioral state B_l , we can determine whether r_{kl} is non-zero. For instance, in the *Controller* component from Figure 2b defects of classes 1, 2, and 3 can occur in states B_2 , B_3 , and B_5 , respectively. Thus, we add transitions (with non-zero probabilities) from B_2 to F_1 , from B_3 to F_2 , and from B_5 to F_3 . In this example, recovery from any failure returns the component back to state B_l . The self-transitions at F_1 , F_2 , and F_3 represent the component being in a failure state until recovery is complete.

Of course, knowing which failure (f_{ij}) and recovery (r_{kl}) transition probabilities 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 estimation. We take this approach in Section 4, which allows us to explore how sensitive the component’s reliability is to (a) each of the defect classes and (b) the recovery process of each defect class.

To reduce the design search space, we could again take advantage of the available information sources. For instance, a domain expert could help the reliability modeler determine how difficult it is to recover from a failure due to defect class k . In turn, this would indicate the value ranges for r_{kl} with which we should experiment. Obtaining and interpreting appropriate ranges of such values is a primary objective of our research.

3.3. Phase 3: Computing Reliability

In this phase we estimate 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 the component is in state i at time t , where $i = F_1, \dots, F_M, B_1, \dots, B_N$. As t goes to infinity (i.e., as the component operates for a long time), these probabilities converge to a stationary distribution,

$$\hat{\pi} = [\pi(F_1), \dots, \pi(F_M), \pi(B_1), \dots, \pi(B_N)]$$

which is uniquely determined by the following equations:²

$$\begin{aligned} \sum_{i \in S} \pi(i) &= 1 \\ \hat{\pi} &= \hat{\pi}P \end{aligned} \quad (1)$$

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, i.e.,

$$R = 1 - \sum_{i=1}^M \pi(F_i) \quad (2)$$

1. Note that we have assumed here that a component will recover from a failure due to one defect before experiencing a failure due to another defect. This assumption may not be reasonable in the case of multi-threaded components (whose dynamic behaviors would comprise multiple concurrently executing state machines). We currently treat such complex components as systems in their own right and apply our system-level reliability prediction technique on them [20].
2. It is not difficult to show that for our reliability model this limiting distribution exists and is a stationary one [21].

As an illustration, assume that in our example from Figure 2b the non-zero failure probabilities are $f_{21}=0.05, f_{32} = 0.04, f_{53} = 0.02$, and that the non-zero recovery probabilities are $r_{11} = 0.2, r_{21} = 0.8, r_{31} = 0.6$. These values were obtained from SCRover’s chief developer. This gives us the following P :

$$\begin{bmatrix} 0.8 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0.8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.4 & 0.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0.05 & 0 & 0 & 0.0076 & 0 & 0.0855 & 0.8569 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0 & 0 & 0.96 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0.02 & 0.0059 & 0.9741 & 0 & 0 & 0 \end{bmatrix} \quad (3)$$

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

After solving Equation (1), we have

$$\begin{aligned} \hat{\pi} &= [\pi(F_1), \pi(F_2), \pi(F_3), \pi(S_1), \pi(S_2), \pi(S_3), \pi(S_4), \pi(S_5)] \\ &= [0.0769 \ 0.0013 \ 0.0096 \ 0.0262 \ 0.3075 \ 0.0263 \ 0.2635 \ 0.2887] \end{aligned}$$

Thus, the reliability of the *Controller* component is estimated to be $R = 1 - (0.0769 + 0.0013 + 0.0096) = 0.9122$

While we reiterate that the goal of our technique is not to provide exact reliability values, those values can be highly accurate in certain situations. In this case, we had access to detailed information about the SCRover system and its *Controller* component. This allowed us to obtain a very accurate reliability estimate in comparison to the implemented component’s reliability, which was calculated to be 0.9257.

4. EVALUATION

In this section, we present the results of evaluating our component reliability prediction framework. Since our framework is primarily intended to be used at system design time, a direct comparison of reliability numbers predicted by the framework and those measured at runtime would not be meaningful. Design-time approaches such as ours are intended for relative comparisons between possible fault mitigation choices rather than (literally) accurate reliability number 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 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 the results obtained using architectural 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 obtain or develop from scratch. Examples include components from

- a cruise control system [18];
- the SCRover robotic testbed, 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 collaboration 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 graduate software engineering project course over the past decade [25].

In order to observe the trends in our framework’s reliability predictions 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 evaluations 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 sensitivity, we will use as an example a component from the DeSi environment whose size and complexity is representative of many software components.

DeSi [13] is an environment that supports specification, manipulation, and visualization of deployment architectures for large-scale, highly distributed systems. It is a stand-alone system, and has also been integrated with MIDAS [12]. DeSi allows an architect to enter desired system parameters into the model, and also to manipulate those parameters and study their effects. DeSi consists of three major subsystems: a reactive *DeSiModel* subsystem that stores information about the current deployment; a *DeSiView* subsystem that visualizes information in the *DeSiModel* subsystem; and a *DeSiController* subsystem 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 higher-order components (i.e., DeSi subsystems), and was accompanied with existing architectural models. For consistency, we will show the evaluation results of applying our reliability prediction framework to the *DeSiController* component only. A slightly abridged dynamic behavior model of *DeSiController* is depicted in Figure 3a. 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.

Table 1: Defects injected in *DeSiController*

Defect	Description	Affected State
d_1 (interface)	Interface mismatch with <i>DeSiViewer</i>	Waiting for command
d_2 (behavior)	Missing model validation rules in the design document	Validating model
d_3 (protocol)	Arguments passed to <i>DeSiModel</i> in wrong order	Finished mapping
d_4 (interface)	Interface mismatch with <i>DeSiModel</i>	Starting blank model

To validate our results, we built separately a model from the existing implementation of the *DeSiController* component. This code-level 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, 5]). 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 defect classes other than the one under consideration 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 mod-

els. The results of those experiments showed qualitatively similar trends to the results presented below.

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. 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.

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

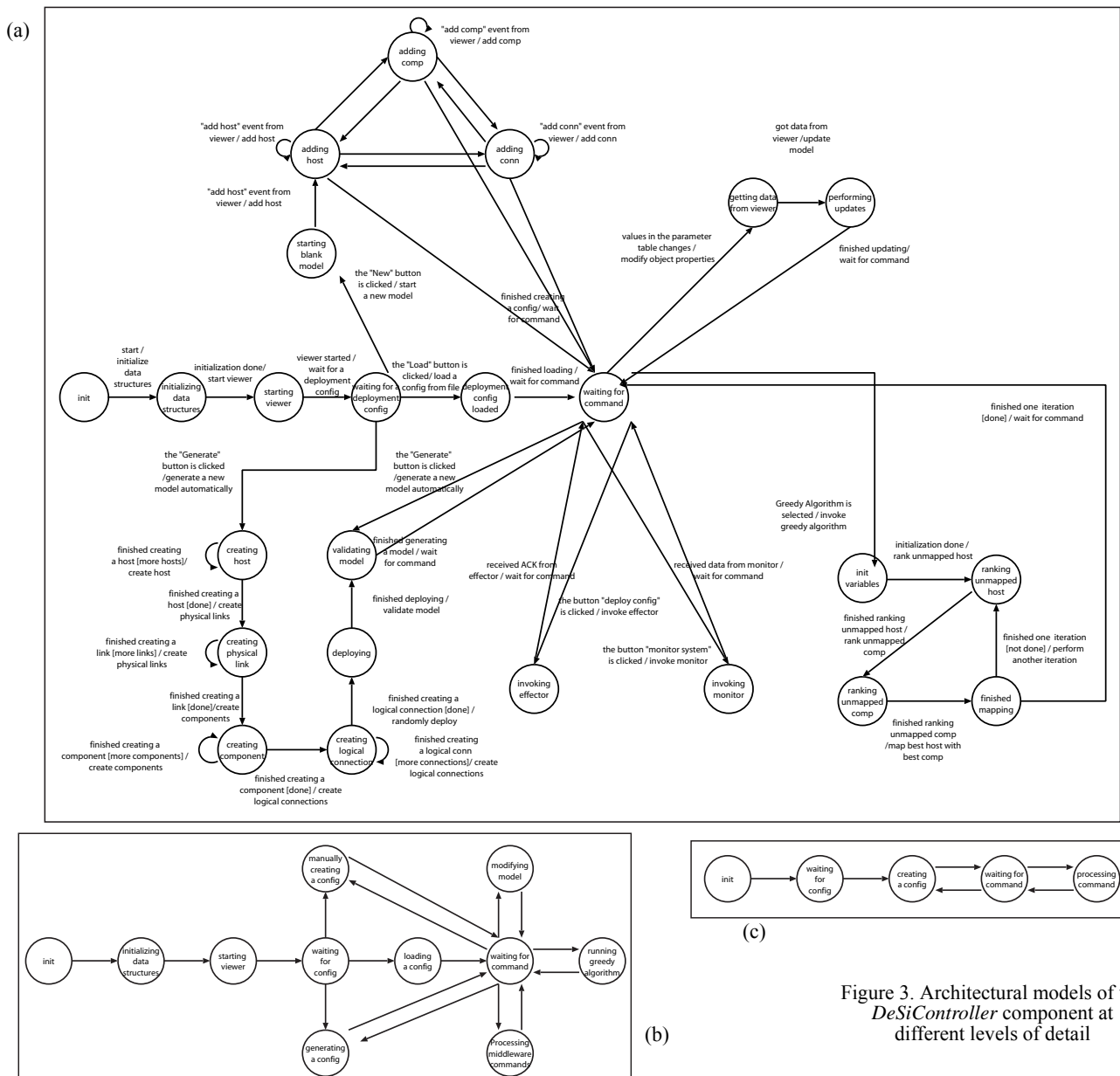


Figure 3. Architectural models of the *DeSiController* component at different levels of detail

- 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 3a. We obtained training data by leveraging the simulation trace and applied our HMM-based approach to obtain behavioral transition probabilities (recall Section 3.2).
- Functionally similar component – We obtained training data from an older version of DeSi that was missing certain functionality (e.g., generating a deployment plan based on a given set of constraints). We again applied our HMM-based approach to obtain behavioral transition probabilities.

Our results are presented in Figure 4, 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 (again, corresponding to the defect class under consideration). Specifically, we activated defect d_1 from Table 1 in Figure 4a, defect d_2 in Figure 4b, defect d_3 in Figure 4c, and defect d_4 in Figure 4d. We observe that the trends conform to our expectations in all four cases for all defects: as recovery probability increases, the reliability of the component increases since the time taken to recover from a failure is reduced. 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 since failure probability is not zero and recovery from a failure is not instantaneous.

Though the general trends in the values across the experiments are similar, Figure 4 yields some interesting observations. First, the sensitivity of the Case (1) (Domain Expert) results, and their accuracy as compared to the Code-level Model results, varies depending on the defect being studied (e.g., compare the results in Figures 4a and 4d). We have observed this situation in a number of other examples we have studied. This indicates that information provided by an expert may be inaccurate, or that the component does not behave as expected. Therefore, solely relying on domain knowledge provided by an expert may produce inaccurate results.

Another observation is that in Figure 4b, reliabilities in Case (3) (Functionally similar component) are very high. This is because

the older 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 4d, 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 in both components. Information from other sources will be required to predict the effect of newly added functionality of certain defect classes.

We also note that in the experiments of Figure 4, the Code-level Model exhibits higher reliability than the other cases. The explanation for this is similar to the one given in Section 4.3 in the context of different granularity models (i.e., the Code-level Model is much finer-grained than the architectural models). As explained below, this is a desirable property of our 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).

4.2. Sensitivity to Operational Profile

To illustrate our reliability framework’s sensitivity to changes in a component’s operational profile, we 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 framework’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 3a) varies from 0 to 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 should reiterate that the same analysis was performed by varying transition probabilities between other states, and yielded quali-

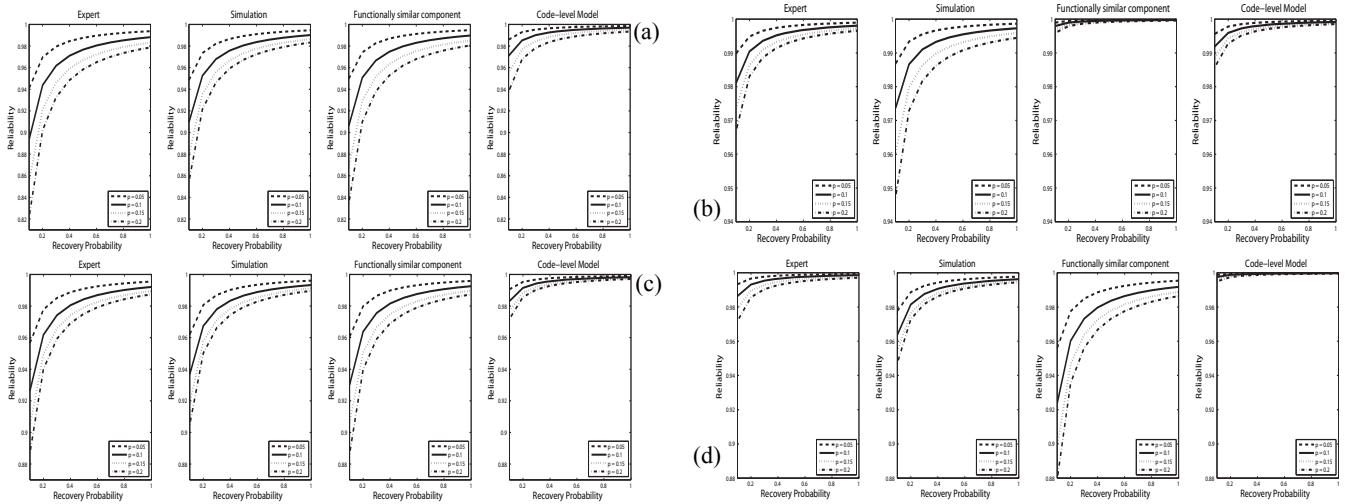


Figure 4. Sensitivity analysis of information sources

tatively similar results. We varied both the failure and recovery probabilities (as in Section 4.1), and obtained a reliability range for each failure and recovery probability pair. We did this for all four defects from Table 1.

Figure 5 depicts our results. Each graph in this figure represents a case with a given failure (fp) and recovery (rp) probability. In each graph, 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* from 0 to 1. The bars labeled (i), (ii), (iii), and (iv) represent the defects d_1 , d_2 , d_3 , and d_4 , 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 5. 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 d_1 and d_3 . This is because d_1 and d_3 directly affect the two states on which we focused in this particular scenario. More generally, by varying operational profiles, we can identify which defects factor most prominently, across these operational profiles, in the resulting reliability values. 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 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, it is questionable how representative a compact high-level model is of the potentially very large and very complex system it is intended to represent. Also, as we will elaborate below, a high-level model may obscure defects that can easily creep in during design refinement and implementation.

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 6 shows the results of calculating the reliability of the *DeSiController* component based on its models at three different levels of granularity, with injected defect d_3 from Table 1, and its operational profile estimated by the

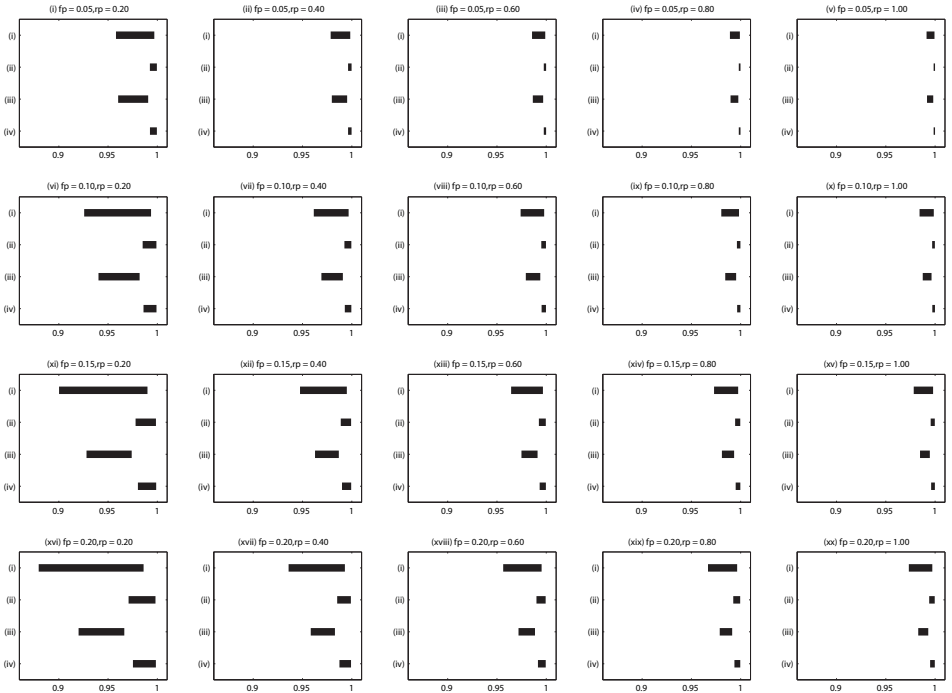


Figure 5. Sensitivity analysis of operational profiles

DeSi expert (recall Case (1) in Section 4.1). Here again we plot reliability as a function of recovery probability from d_3 -related failure, and the different curves correspond to failure probabilities due to d_3 . Performing this analysis using other information sources (functionally similar component and simulation) and other defects consistently yielded qualitatively similar results, and we omit them due to lack of space.

The detailed model of *DeSiController* from Figure 3a 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 3b and 3c. 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 3b abstracts away the portion of the state machine comprising the four states and their transitions in the lower right segment of Figure 3a.

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 3c). Another observation is that the model from Figure 3b is more sensitive to recovery probability than the model from Figure 3c, while the most detailed model (Figure 3a) is least sensitive. This can be explained as follows. Failures corresponding to defect d_3 only take place in the *Finished mapping* state of the 24-state model (Figure 3a). On the other hand, time spent in the *Running algorithm* state in the 11-state model (Figure 3b) 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, comparatively more time is spent in the *Running algorithm* state in the 11-state model than in the 24-state model, hence the sensitivity is higher in the 11-state model. Likewise, since the time spent in the *Processing command* state in the 5-state model (Figure 3c) includes the

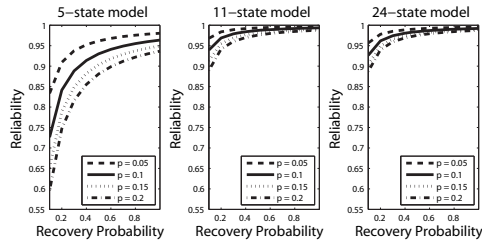


Figure 6. Sensitivity analysis of models of different granularities

time spent in *Running algorithm*, *Modifying config*, and *Processing middleware command* states in the 11-state model, the sensitivity of the 5-state model is higher than that of the 11-state model.

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 3b) 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 estimate of a component’s reliability (i.e., smaller models require less computation). Specifically, in our experiments, a model with fewer states gives more pessimistic results. In general, this is a desirable property for a reliability estimation framework, i.e., to give more conservative reliability estimates given less information.

5. CONCLUSIONS

Meaningful architecture-level reliability estimation is critical to the cost-effective development of complex software systems. However, the uncertainties that exist in the early stages of software development (e.g., lack of knowledge about operational profiles) pose significant challenges to the development of architecture-level reliability estimation techniques. Early works in this area, specifically in the context of component-based systems, all have assumed some degree of knowledge of components’ reliabilities. In this paper, we have argued that these assumptions are not reasonable, and have presented a framework for component-level reliability estimation that does not rely on such assumptions.

We approached the challenges associated with the above mentioned uncertainties by (a) exploiting behavioral models which 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 which have been successfully applied in dependability modeling of hardware systems as well as in other engineering fields.

Our evaluation results indicate that our framework provides meaningful reliability prediction in the context of early stages of software development. Our on-going and future research directions focus on extending our framework to system-level reliability estimation [20] and on doing so in the context of a variety of development scenarios. We expect the results presented in this paper to become an integral part of our system-level estimation techniques. At the same time, we believe that the scalability of reliability estimation techniques is an additional challenge which we must address in developing system-level reliability prediction approaches.

6. REFERENCES

- [1] Boehm B., et al. Using Empirical Testbeds to Accelerate Technology Maturity and Transition: The SCRover Experience, in *Proceedings of ISESE’04*, pp. 117 - 126, 2004.
- [2] Cheung, R.C. A User-Oriented Software Reliability Model, *IEEE Transactions on Software Engineering*, 6 (2), 1980.
- [3] Goel A.L., Okumoto K. Time-Dependent Error-Detection Rate Models for Software Reliability and Other Performance Measures, *IEEE Trans. on Reliability*, 28(3):206–211, 1979.
- [4] Gokhale, et al. Reliability Simulation of Component Based Software Systems, *Proceedings of ISSRE’98*, pp. 192-201, 1998.
- [5] Goseva-Popstojanova K. et al. Architectural Level Risk Analysis using UML, *IEEE Transactions on Software Engineering*, Vol.29, No.10, October 2003.
- [6] Goseva-Popstojanova K., Kamavaram S. Assessing Uncertainty in Reliability of Component-Based Software Systems, in *ISSRE 2003*, Denver, CO, Nov. 2003.
- [7] Goseva-Popstojanova K. et al., Architecture-Based Approaches to Software Reliability Prediction, *International Journal Computer & Mathematics with Applications*, Vol. 46, Issue 7, October 2003.
- [8] Jelinski, Z. and Moranda, P. B., *Software Reliability Research, Statistical Computer Performance Evaluation*, edited by W. Freigerger, Academic Press, 1972.
- [9] Littlewood, B., Software Reliability Model for Modular Program Structure, *IEEE Transactions on Reliability*, 28 (3), 1979.
- [10] Littlewood, B.A., Verrall, J.L., A Bayesian Reliability Growth Model for Computer Software, *Applied Statistics*, Vol. 22, 1973.
- [11] Lyu, M.R (Editor). *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, NY, 1996.
- [12] Malek S., et al., Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. To appear in the *Proceedings of ICSE 2007*, May 2007.
- [13] Malek S., et al. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In R. de Lemos, C. Gacek, and A. Romanowski, eds., *Architecting Dependable Systems III*, Springer Verlag, October 2005.
- [14] Perry, D.E., and Wolf, A.L. Foundations for the Study of Software Architecture, *Software Engineering Notes*, 17(4), 1992.
- [15] Rabiner L.R., A Tutorial on Hidden Markov Models, in *Proceedings of the IEEE*, vol. 77, pp. 257-286, 1989
- [16] Reussner R., et al. Reliability prediction for component-based software architectures, *J. of Systems and Software*, 66(3), 2003.
- [17] Roshandel R., et al. Understanding Tradeoffs among Different Architectural Modeling Approaches. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway, June 2004.
- [18] Roshandel R., et al. Estimating Software Component Reliability by Leveraging Architectural Models. *Emerging Results track, ICSE 2006*, pp. 853-856, Shanghai, China, May 2006.
- [19] Roshandel R., Medvidovic N. Multi-View Software Component Modeling for Dependability, in *Architecting Dependable Systems II*, LNCS, 2004.
- [20] Roshandel R. et al. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. Under review.
- [21] Stewart W.J., *Introduction to the numerical solution of Markov Chains*. Princeton University Press, Princeton, NJ, 1994.
- [22] Shaw M., Garlan D., *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, Inc., 1996.
- [23] Shooman, M., Structural Models for Software Reliability Prediction, in *Proceedings of ICSE’76*, pp. 268-280, 1976.
- [24] Wang W., Wu Y., Chen M., An architecture-based software reliability model, in *Proc. of Pacific Rim International Symposium on Dependable Computing*, 1999.
- [25] <http://greenbay.usc.edu/csci577/>, CSCI577 – Software Engineering, University of Southern California.