DESIGN-TIME SOFTWARE QUALITY MODELING AND ANALYSIS OF DISTRIBUTED SOFTWARE-INTENSIVE SYSTEMS

by

Leslie Chi-Keung Cheung

A Dissertation Presented to the

FACULTY OF THE USC GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCE)

May 2011

Copyright 2011 Leslie Chi-Keung Cheung

Acknowledgements

First of all, I would like to express my thanks to my advisor Leana Golubchik. Without her guidance I would not have completed this dissertation On technical matters, she gave me a lot of useful comments to improve this dissertation. Personally, she gave me tremendous amount of support and encouragement, especially during some very hard times.

Thanks also go to Nenad Medvidovic, who has helped me significantly during my PhD journey. Neno has helped me with his software engineering wisdom, and has always been available when I needed help.

I thank Sandeep Gupta, Gaurav Sukhatme and Shahram Ghandeharizadeh for serving on my guidance committee, and for providing invaluable feedback to my work. Also, I thank William GJ Halfond and Fei Sha for their comments on improving an earlier version of the Web service work.

I would like to take this opportunity to thank my first research advisor, Andrea Arpaci-Dusseau of UW-Madison, for introducing me to computer systems research. I would not have thought about doing a PhD without the enjoyable experience working with her.

I am grateful to have an opportunity to work with my colleagues in the Internet Multimedia Lab: Alix Chow, Yan Yang, Yuan Yao, Kai Song, Bo-Chun Wang, Sung-Han Lin, Ranjan Pal and Abhishek Sharma. I thank each of you for your feedback on

my work, and for staying for my long and endless presentation dry-runs. I enjoy my time working with each of you.

I appreciate all the help I got from my colleagues in the software architecture group. Ivo Krka deserves special thanks, as he has helped me tremendously in improving SHARP, as well as reading early drafts of my papers and this dissertation. I also thank Roshanak Roshandel for the fruitful discussions on the component operational profile estimation work, Sam Malek for his help on the DeSi experiments, and George Edwards and Chiyoung Seo for their help on the MIDAS experiments.

Finally, I cannot imagine completing my PhD without the love and support from my family. My parents, Alex and Susanna, have been my huge supporters, and helped me in any way they can. I always enjoy spending my time with my brothers, Frank and Felix, whenever we can. Felix, I wish you good luck on your own PhD journey.

Contents

Acknowledgements **ii**

List of Tables **vi**

List of Figures **vii**

Abstract **x**

1 Introduction **1**
1.1 SHARP: A Scalable, Hierarchical, Architecture-Level Reliability Prediction Framework 9
1.2 An Approach to Performance Estimation of Third-Party Web Services from a Client's Perspective 11
1.3 Design-Time Operational Profile Estimation 13
1.4 Contributions and Validation 15
1.5 Roadmap 16

2 Related Work **17**
2.1 Design-Level Software Reliability Analysis 17
2.1.1 Applicability to Concurrent Systems 19
2.1.2 Parameter Estimation 22
2.2 Testing-Based Software Performance Estimation Techniques 23

3 SHARP: A Scalable Framework for Reliability Prediction of Concurrent Systems **27**
3.1 Background 27
3.1.1 Architectural-Level Defect Analysis 31
3.2 An Overview of the SHARP framework 32
3.3 Reliability Computation 35
3.3.1 Basic Scenarios 36
3.3.2 SEQ Scenarios 44
3.3.3 PAR Scenarios 48

3.4 Evaluation . . . 58
3.4.1 Complexity Analysis . . . 60
3.4.2 Accuracy . . . 66
3.5 Conclusions . . . 71

4 Performance Estimation of Third-Party Components 72
4.1 A Framework for WS Performance Prediction . . . 73
4.1.1 Step 1: Performance Testing . . . 74
4.1.2 Step 2: Regression Analysis . . . 75
4.2 Validation . . . 89
4.2.1 Interpolation Errors . . . 90
4.2.2 Extrapolation Errors . . . 95
4.3 Conclusion . . . 99

5 Parameter Estimation in Quality Analysis of Software Components 100
5.1 Component Reliability Prediction Framework . . . 101
5.2 Operational Profile Modeling . . . 105
5.3 Evaluation of Operational Profile Estimation . . . 111
5.3.1 Evaluation of SCRover's Controller . . . 113
5.3.2 Evaluation of DeSi . . . 120
5.4 Conclusions . . . 128

6 Conclusions and Future Work 130
6.1 Summary and Contributions . . . 130
6.2 Future Work . . . 132
6.2.1 Integrating Firmware Properties . . . 132
6.2.2 Performability Analysis . . . 133
6.2.3 Reliability Testing . . . 134

Bibliography 136

List of Tables

3.1 r_i and t_i of the MIDAS scenarios . . . 44

3.2 Values of $P(C_k)$ and $R(C_k)$ in the *System* scenario . . . 50

3.3 Values of $P_j(c_j)$ in the *System* scenario . . . 54

3.4 Worst-case complexities . . . 61

3.5 Summary of computational costs in practice . . . 64

4.1 Comparisons of TPC WS interpolation results . . . 86

4.2 Errors in response time estimates using QN^3 . . . 89

4.3 TPC WS interpolation errors . . . 91

4.4 Average Interpolation Errors . . . 91

4.5 Extrapolation Errors . . . 96

5.1 Defects injected in *DeSiController* . . . 122

List of Figures

1.1 The problem space in early software quality analysis — Cost vs. Information Availability . . . 3

1.2 The problem space in early software quality analysis — Quality metric vs. Information Availability . . . 5

2.1 An example of the Cheung's model [20] . . . 17

3.1 An Overview of the MIDAS system . . . 27

3.2 Components' state diagrams . . . 28

3.3 Sequence diagrams . . . 29

3.4 MIDAS scenarios organized in a hierarchy . . . 31

3.5 An illustration of SHARP applied on the complex *Sensor measurement* scenario . . . 35

3.6 Component submodels of *SensorGW* . . . 37

3.7 SBMs of the basic scenarios . . . 38

3.8 QN model of the *SensorGW* scenario . . . 41

3.9 Rate redistribution in *GUIRequest* . . . 43

3.10 SBMs of the SEQ scenarios . . . 46

3.11 Models for completion rate computation for *GUI_LOOP* and *ControlAC* 47

3.12 SBMs of the PAR scenarios . . . 51

3.13 Probability distribution of the number of completed instances . . . 52

3.14 Computational Cost in Practice . . . 65

3.15 Sensitivity Analysis of the *Sensor_PAR* scenario . . . 67

3.16 Sensitivity analysis at the system level 68

3.17 Sensitivity analysis of the Client-Server system 69

3.18 Computational cost of SHARP with and without truncation 70

3.19 Errors caused by model truncation 70

4.1 An overview of our WS performance prediction framework 72

4.2 Extrapolation using standard regression methods 78

4.3 An Example Objective Function 80

4.4 Extrapolation using queueing models 81

4.5 TPC WS Architecture 82

4.6 An overview of the hybrid approach 87

4.7 Results using QN^3 88

4.8 Interpolation 92

4.9 Extrapolation 97

5.1 Dynamic Behavior Model of the *Controller* Component 101

5.2 Software Component Reliability Prediction Framework 102

5.3 Reliability Model of the *Controller* Component 103

5.4 Analysis of sensitivity to information sources of SCRover's *Controller* . 115

5.5 Analysis of sensitivity to operational profiles of SCRover's *Controller* . 117

5.6 Dynamic behavior models of the *Controller* component at two different levels of granularity. 119

5.7 Analysis of sensitivity to models of different granularities of SCRover's *Controller* 120

5.8 Architectural models of the *DeSiController* component at different levels of detail 121

5.9 Analysis of sensitivity to information sources of *DeSiController* 124

5.10 Analysis of sensitivity to operational profiles of *DeSiController* 126

5.11 Analysis of sensitivity to models of different granularities of *DeSiController* 127

Abstract

As our reliance on software system grows, it is becoming more important to understand a system's quality, because systems that provide poor quality of service have costly consequences. It has been shown that addressing problems late, such as after implementation, is prohibitively expensive, because it may involve redesigning and reimplementing the software system. Thus, it is important to analyze software system quality early, such as during system design. In early software quality analysis, in addition to analyzing components that are developed from scratch, it is also necessary to analyze existing components that are being integrated into the system, because software designers make use of them to save development cost.

We focus on two aspects of early software quality analysis: the cost of analysis and parameter estimation. First, we address the high cost of existing design-level quality analysis techniques. In modeling complex systems, existing design-level approaches may generate models that are computationally too expensive to solve. This problem is exacerbated in concurrent systems, as existing design-level approaches suffer from the state explosion problem. To address this challenge, we propose SHARP, a design-level reliability prediction framework that analyzes complex specifications of concurrent systems. SHARP analyzes a hierarchical scenario-based specification of system behavior and achieves scalability by utilizing the scenario relations embodied in this hierarchy. SHARP first constructs and solves models of the basic scenarios, and combines the

obtained results based on the defined scenario dependencies; this process iteratively continues through the specified scenario hierarchy until finally obtaining the system reliability. Our evaluations indicate that (a) SHARP is almost as accurate as a traditional non-hierarchical method, and (b) SHARP is more scalable than other existing techniques.

Second, we address the high cost of testing-based approaches, which are typically used in analyzing the quality of existing software components. However, since testing-based approaches require sending a large number of requests to the components under testing, it is quite an expensive process, particularly when testing at high workloads (i.e., where performance degradations are likely to occur) — this may render the component under testing unusable during the tests' duration (which is also a particularly bad time to have a system be unavailable). Avoiding testing at high workloads by extrapolating (from data collected at low workloads), e.g., through regression analysis, results in lack of accuracy. To address this challenge, we propose a framework that utilizes the benefits of queueing models to guide the extrapolation process, while maintaining accuracy. Our extensive experiments show that our approach gives accurate results as compared to standard techniques (i.e., use of regression analysis alone).

Finally, we address the problem of parameter estimation in existing design-level approaches. An important step in software quality analysis is to estimate the model parameters, which describe, for example, how the system and its components are used (this is known as their operational profile). This information is assumed to be available in existing design-level approaches, but it is unclear how existing approaches obtain such information to estimate model parameters. We identify sources of information available during design, and describe how information from different sources can be translated for use in the context of component reliability estimation. Our evaluation and validation

experiments indicate that use of our approach in determining operational profiles results in accurate reliability estimates, where implementations are used as ground truth.

Chapter 1

Introduction

Software systems play a major role in our everyday lives. Nowadays, we rely on software systems to perform many tasks, including personal communication using email and instant messaging, business applications for online shopping and business-to-business services, software controllers of medical devices, aircraft control systems, and so on. As our reliance on software systems grows, analyzing the system's quality has become more important. Software systems providing poor quality of service may cause inconvenience, hurt business income and reputation, and may even cause loss of human lifes.

Traditionally, software system quality system is after the system has been built using testing-based approaches. For example, to ensure the correctness of a system, software engineers prepare a suite of test cases, with a variety of valid and invalid requests, and compare the system's output with the anticipated output, determined using the system's specifications. Another example is performance testing, for ensuring that the system provides acceptable performance. In a performance test, software engineers generate a large number of requests according to some traffic models or existing workloads, and measure various system performance metrics, such as system throughput, average response time, and utilization. However, testing-based approaches are expensive, because they involve sending a large number of requests to ensure the system is tested thoroughly. In many cases, correcting the problems can be even more expensive, as mitigating the problems may involve redesigning, rebuilding, and retesting the entire system, which can be orders of magnitude more expensive than if such problems are discovered and addressed during system design [14].

At the same time, during system design, software engineers are faced with many design decisions, many of which have significant impact on software quality. For example, if the software system is designed to be deployed in harsh physical environments (e.g., in a tropical rain forest), system designers need to consider the level of redundancy needed for their applications: this requires studying the tradeoff between cost and system reliability in deploying redundant components. As another example, software engineers may choose between developing a software component from scratch, or utilizing a third party component. The use of third-party components saves development cost, but integrating the component into the system under design may not be always trivial, and it may be more difficult to address problems when they arise. Therefore, *early* quality analysis, such as during software architecture design, is important in building high quality software systems. Software architecture provides high-level abstractions for representing the structure, behavior, and key properties of a software system [53, 71]. A software system's architecture comprises a set of computational elements (components), their interactions (connectors), and their compositions into systems (configurations). Early software quality analysis allows software designers to study design tradeoffs and assess their impacts on software quality, such that software designers can make more informed design decisions, and hence architect better software systems.

We focus on two important aspects in early software quality analysis: the *cost of the analysis* and *parameter estimation*. Before discussing these in details, let us consider the problem space in early software quality analysis, which is depicted in Figure 1.1. The x-axis describes the amount of information available to system components. Software engineers may design new components or integrate existing components, possibly provided by a third-party, into the system. Information available about existing components varies: making use of existing components that are developed in-house (e.g., from an older version of the system) or open-source software allows access to the source code

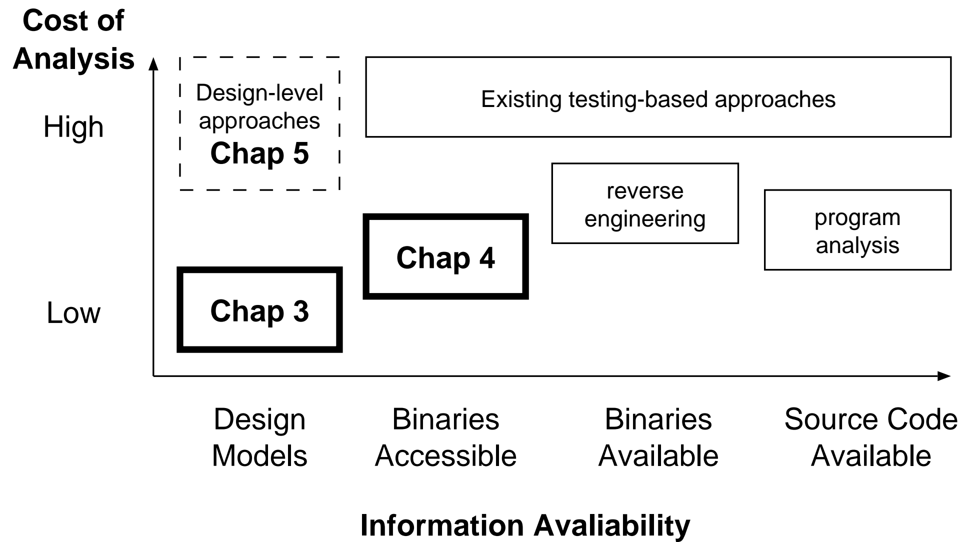


Figure 1.1: The problem space in early software quality analysis — Cost vs. Information Availability

and perhaps expert information about the components (Source Code Available); purchasing software from third-party vendors may allow access to only the binaries (Binaries Available); and, utilizing software that is deployed by a third-party allows access to the component's services, but neither the source code nor the binaries are available as the component is deployed by a third-party (Binaries Accessible). In the cases above, we can apply testing-based techniques as the component has been built. On the other hand, when software engineers design new components, only design models are available for software quality analysis (Design Models). Testing-based techniques are not applicable, because the component has not yet been built.

The y-axis in Figure 1.1 describes the cost of analysis. While testing-based techniques can be applied when the implementation is available, they are expensive as they involve making a large number of requests. If the source code or the binaries are available, program analysis- and/or reverse engineering-based techniques can be applied. While these techniques have lower cost, they are not applicable when the binaries are

accessible but unavailable for quality analysis (i.e., software in the “Binaries Accessible” category). For example, reverse engineering-based approaches cannot be applied when the software is hosted by a third party, in which users have accesses to the service, but cannot obtain the binaries. While a number of approaches that leverage design models have been proposed (when only design models are available) (see Chapter 2 for details), they are costly to apply. As the scale and degree of concurrency of modern software systems have grown significantly, incorporating the complex relationships between different parts of the system in a tractable way has become more challenging. An intractable approach would be too expensive to apply, and hence not useful in evaluating and improving the system. Thus, the computational cost of solving for the quality metric of interest is prohibitively high in existing design-level approaches for larger systems, and this scalability problem is exacerbated in concurrent systems, in which the state space is much larger.

Moreover, none of the existing design-level approaches discusses how model parameters, which describe the software system runtime behaviors, can be obtained. These approaches are not useful without accurate estimation of the model parameters. This is illustrated as a dotted box in Figure 1.1.

Figure 1.2 shows another dimension of the problem space. The y-axis represents software quality metrics, that each metric describes a different aspect of quality of the system. In this dissertation, we focus on analyzing system *performance* and *reliability*. Performance usually refers to the response time or throughput as seen by the users [67], and reliability can be informally defined as the probability that the system performs “correctly”, as specified in its requirements specification.

When the source code is available, program analysis-based techniques can be applied to evaluate the system’s performance and reliability. For example, software profiling techniques (e.g., [27]) identify how much time is spent on different parts of

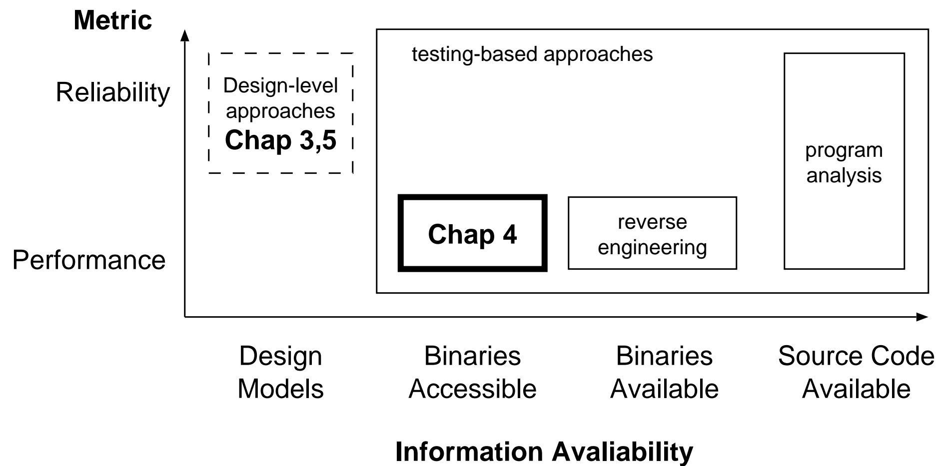


Figure 1.2: The problem space in early software quality analysis — Quality metric vs. Information Availability

the code, while code-level reliability analysis techniques (e.g., [20]) build a reliability model from the source code, and solve the model for a reliability estimate.

When the source code is unavailable, although reverse engineering-based approaches (e.g., [40]) have been applied in performance estimation in the "Binaries Available" case, it is typical to rely on testing-based approaches to evaluate a system's performance and reliability. For example, we send a large amount of requests and measure the average response time for performance analysis, and observe the number of errors the system returns for reliability analysis. As noted earlier, the cost of testing-based approaches is high, and we strive to reduce their cost in this dissertation.

In Chapter 3, we address the high cost of design-time reliability analysis. We choose to tackle this problem because (1) during system design, it is imperative to ensure the system is reliable, or the system would not be usable; and (2) existing design-level approaches are unable to scale to larger systems. We focus on reliability instead of performance here because reliability can be defined more broadly to include performance characteristics as well, by specifying performance requirements in the requirements

specification. (Recall that a system is considered unreliable when it violates any requirement documented in its requirements specification) For instance, a system designer may specify a requirement that the system should process a time-sensitive request within X seconds. If the system fails to process such request within the specified time, it is considered unreliable. As part of our future work, we plan to integrate performance and reliability into one unified framework, which is typically known as performability [50] (see Chapter 6.2.2).

In Chapter 4, we focus on reducing the cost of testing-based approaches in evaluating the performance of software in the "Binaries Accessible" category. We choose to tackle this problem because (1) performance testing is expensive, especially at high workload; and (2) there is no alternative to testing-based approaches to evaluate software this category. Reliability testing is also very expensive, as it involves sending a very large number of requests to observe an error. For example, a system's reliability is usually specified using the five 9's rule. i.e., its reliability is expected to be at least 99.999%. This implies that, on average, it requires sending 100,000 requests to this system before we observe an error. Reducing the cost of reliability testing remains a challenge; we consider our work in Chapter 4 in reducing the cost of performance testing as a first step in this direction, and consider reliability testing as part of our future work (see Chapter 6.2.3).

In Chapter 5, we address the problem of parameter estimation in design-level reliability analysis. This is an important problem because it is unreasonable to assume the availability of an "oracle" to provide model parameters, as they typically correspond to the system's runtime behavior. In addition, we will show, in Chapter 5.3, that even if such an oracle exists, the information may be inaccurate, which results in inaccurate reliability estimates. Parameter estimation in design-level performance estimation

approaches is also an important topic. However, unlike reliability estimation, performance estimation requires information from the underlying platform, and integrating such information during system design is challenging. We address this problem as part of our future work (see Chapter 6.2.1). We note that system reliability is also affected by the reliability of its underlying platform. Yet, many reliability problems are rooted in design errors, and these problems manifest themselves regardless of which platform we deploy the software on. Integrating firmware properties into our reliability analysis is also part of our future work.

We approach these problems as follows:

1. We address the high cost of design-level approaches in the "Design Models" category in Figure 1.1 by proposing SHARP (Chapter 3). SHARP is a design-level reliability prediction framework that has significantly lower computational cost than existing approaches. More specifically, SHARP analyzes a small part of the system at a time, according to the system *use-case scenarios*, which is a standard way software engineers divide a system into smaller pieces. The results are then combined to obtain a system reliability estimation using a hierarchical solution technique we describe in Chapter 3. The motivation behind SHARP is that solving many smaller models is significantly less expensive than solving one huge model in terms of computational cost.

2. We address the high cost of testing-based approaches by proposing a framework for estimating system performance at high workload intensity, using performance information collected at low workload intensity, and applying regression-based analysis (Chapter 4). More specifically, estimating system performance at high workload intensity is very expensive, as it involves generating a large number of requests. This process may saturate the system under testing, rendering it not

usable. We propose a framework that leverages regression analysis to predict system performance at high workload intensity, by using performance information at low workload intensity collected through testing, which is less expensive to collect. We have applied this technique in predicting the performance of third-party Web services, which correspond to the "Binaries Accessible" category in Figure 1.1, because, as discussed earlier, there is no alternative to estimating performance of software in this category other than through testing-based approaches.

3. To remove the assumption that model parameters are available in existing design-level approaches, we explore the sources of information during design, and study how such information can be used to estimate model parameters in the context of reliability estimation in Chapter 5. Specifically, important information 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. We discuss how we estimate candidate operational profiles for reliability estimation by leveraging and combining information from different sources, and applying the hidden Markov model (HMM)-based approach in [19] to estimate operational profiles using such information.

The remainder of this chapter is organized as follows: we discuss SHARP in more detail in Chapter 1.1; performance estimation of third-party components in Chapter 1.2; and parameter estimation at the design level in Chapter 1.3. Chapter 1.4 presents the contributions of this dissertation. Finally, a roadmap of the remainder of this dissertation is given in Chapter 1.5.

1.1 SHARP: A Scalable, Hierarchical, Architecture-Level Reliability Prediction Framework

In a nutshell, existing design-level approaches generate a performance or a reliability model from the software system's architecture, which takes information about component interactions and the performance or reliability of individual components as parameters. The metric of interest is computed by solving the system-level model. Several survey papers have been published in this area. For example, [10, 12] are surveys on early performance analysis, and the surveys in [34, 38, 30] discuss early reliability prediction. For example, the performance modeling approach in [22] generates an execution graph to describe component interactions. Then, it generates a queueing network from the execution graph and the system's deployment plan, which describes on which host each component is deployed.

In combining component models to compute a system reliability estimate, existing approaches have not considered the issue of scalability of the reliability model and its solution. i.e., they may result in intractable models for larger systems. This is especially the case in reliability prediction of concurrent systems, in which it is typical (e.g., as in [28, 59]) to keep track of the status of all components. The size of the model is $O(\mathbf{M}^{\mathbf{C}})$, where $\mathbf{M}$ is the number of states in a component, and $\mathbf{C}$ is the number of components. That is, the number of states grows exponentially with the number of components, causing the so-called state space explosion problem, which makes the model solution intractable.

To address the problem of *scalable* reliability prediction of concurrent systems, in Chapter 3, we propose SHARP, a hierarchical reliability prediction framework. In SHARP, rather than considering a concurrent system as having simultaneously running components as in existing approaches (such as in [59]), we view it as having different

use-case scenarios that execute concurrently. For example, consider a sensor network application where a number of sensors take measurements and users can read the processed data at a GUI. We view it as having a sensor measurement scenario and a GUI display scenario running simultaneously. SHARP is also capable of handling complex scenarios, in which a scenario is composed of several lower-level scenarios. The lower-level scenarios may be complex scenario themselves, and SHARP is capable of handling complex scenarios with an arbitrary number of levels.

As inputs to our framework, we require the system use-case scenario models (e.g., UML sequence diagram), a description of how scenarios interact (e.g., Scenario 2 starts after the execution of Scenario 1), and the system's operational profile (estimates of which we discuss in detail in Chapter 5). We also need to identify the defects that cause system failures; we leverage the approach in [63] to identify mismatches between architectural models of the system's component. Our framework produces system reliability estimates, as well as the reliability of each basic scenario, as output. To generate a system model, first we generate models of the basic scenarios by leveraging system use-case scenario models. Then, we combine the models of the basic scenarios to form a higher-level model, according to the relationships between the lower-level scenarios. Thus, system reliability is the reliability of the highest-level scenario. The motivation here is that a model of a scenario is expected to be relatively small, and that solving a number of smaller submodels (rather than one huge model) results in space and computational savings. We note that the use of scenario-based models is also explored in [35, 59, 78]. However, these works differ from SHARP in that [35, 78] assume a sequential system, while [59] considers a "flat model" and thus suffers from the very scalability problem we are striving to address.

We are able to achieve better scalability without sacrificing the level of system detail we can model. More specifically, in modeling concurrent systems, some existing works

(e.g., [28]) model a component as being either on or off. By doing so, while we know which component has failed, it is very difficult to tell what causes a component failure. In SHARP, by using a hierarchical approach, we are able to retain greater level of detail about the system being modeled, while doing so in a scalable way.

The notion of system failure is different in different operational contexts and usages, and from different perspectives, even within a single system. Therefore, in order to provide architects with a comprehensive analysis approach, a reliability estimation framework should be able to capture different notions of system failure. Failure rules specify the conditions under which the system fails, and are more complex in concurrent systems. Existing approaches designed for sequential systems assume the system fails when the running component fails, and it is not obvious how to incorporate other failure rules into their approaches. Thus, we propose an approach which captures different notions of system failure. To this end, we allow designers to specify conditions under which the system is considered reliable, in terms of the number of failed instances of scenarios. In turn, failure rules determine how we combine the solutions of lower-level scenarios in order to compute the overall system reliability. Moreover, our approach to capturing failure rules can be applied to existing reliability prediction approaches for concurrent systems.

1.2 An Approach to Performance Estimation of Third-Party Web Services from a Client's Perspective

As discussed earlier, software designers may utilize third-party components to reduce development cost, such as reusing components from a previous project, or buying software from a third-party vendor. In the case where the component is deployed by a

third-party, one has to rely on testing-based approaches. However, testing the component at high workload may render it not usable during testing, as serving the testing traffic depletes its resource.

We have chosen to study performance estimation in the Web service paradigm for the following reasons, which falls under the "Binaries Accessible" category in Figure 1.1. We argue that analyzing performance and reliability of third-party Web services (WSs) is more challenging than in other categories (e.g., using open-source software and components that are bought from a third-party) because (1) WSs are only required to publish their interfaces (via WSDL [8]); information about their internal structure (e.g., whether they are deployed on a single host or a cluster server) and external resources (e.g., whether they use a remote database or another WS) they rely on are typically unavailable; and (2) testing-based approaches adversely affect the normal operation of a WS, which is already operational.

Existing work on evaluating the quality of WSs has focused on evaluating WSs from a system administrator's or a designer's perspective. For example, [74] assumes the systems architecture is known and models a WS-based system using a multi-tiered architecture. Other works assume the systems architecture (e.g., how the third-party WSs are connected [75]), and/or the systems parameters (e.g., the amount of I/O time needed to complete a service [46]) are known. We argue that such an assumption is not reasonable in evaluating third-party WSs from a *client's perspective*: it is not clear how such information can be obtained by a client, and the service providers may be reluctant to provide it.

We focus on evaluating the performance of third-party WSs from a client's perspective, and our focus is on average response time estimation. Our major challenge is the lack of information about the target WS. This includes (1) the structure of the WS, as

discussed above, and (2) the parameters of each WS that provides service to complete a client's request.

Our proposed approach makes use of data collected from performance testing [51], which involves sending requests and collecting performance data at low workloads, and applying regression analysis [26] to such data for response time prediction at high workloads. Our experiments have shown that applying standard regression analysis gives poor *extrapolation* results, which, in this context, corresponds to predicting the response time *outside* of the parameters used in performance testing. Therefore, we propose to fit the performance data to queueing models for WS response time prediction. Queueing models have been widely used in performance modeling of computer systems. Thus, we believe they are useful in modeling WS performance as well, and hypothesize that predicting performance using queueing models fitted to performance testing data is more accurate than using standard approaches in the regression literature. However, the *interpolation* results of using queueing models, which corresponds to predicting response time *within* the parameters used in performance testing, are not as good as using standard regression approaches. Hence, we derive a hybrid approach that combines the benefits of using queueing models and standard regression approaches.

1.3 Design-Time Operational Profile Estimation

One common theme across existing design-level approaches is that they assume runtime information about a system or a component is available, which is an unreasonable assumption because the system/component has not been implemented. For example, one important ingredient in performance and reliability modeling is the system's and its components' operational profiles, which describe how the system and its components are used [52]. It appears that existing approaches have assumed the availability of an

"oracle" that can provide model parameters; yet, such an oracle typically does not exist. Even if an oracle is available, as we will show in Chapter 5, this information is subjective and may be inaccurate, due to the complexity of the system, or to unexpected interaction patterns between components.

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 behavior constructed from the architectural model and (3) execution logs of functionally similar systems/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 aforementioned 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 reliability. In thus utilizing architectural models we observe that another important ingredient in reliability prediction is information about a system's or 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 [63, 61] to identify inconsistencies within a component's as well as between components' architectural models. We demonstrate a way to study the effects of different failure modes by exploring the design space. i.e., to vary the failure-related parameters between a range of possible values and observe the resulting effects on the component's reliability prediction.

1.4 Contributions and Validation

To summarize, we present the contributions of this dissertation, as well as an overview of our validation process.

Our first contribution is the SHARP framework, that can accurately predict concurrent system reliability, while significantly reducing the computational cost needed to solve for system reliability, as compared to existing design-level, brute-force type approaches. SHARP achieves scalability through the use of a hierarchical approach, and our solution technique allows us to solve for the reliability of a part of a system at a time, and combine the results of lower-level scenarios appropriately. The motivation is that solving many smaller, scenario-based models is more efficient than solving a model of the entire system. Through extensive experimentation we validate the complexity and accuracy of this approach. Lastly, we note that SHARP is an approximation of the "flat model" (i.e., one that keeps track of the status of all components) used in other techniques. However, we argue that its potential scalability benefits are achieved at the cost of fairly small losses in accuracy.

To address the high cost of testing-based approaches in analyzing the performance of third-party components, specifically WSs, we propose a framework for estimating performance at high workload intensity, using information collected at low workload intensity, and applying regression analysis — this is another contribution of this dissertation. Such an approach allows system designers to evaluate the target WS, for example, for its response time and stability conditions, and can be used in determining how the target WS should be used in designing a new WS. We evaluate the accuracy of our approach by studying its interpolation and extrapolation errors. Our results indicate that our approach is able to overcome the poor extrapolation results while maintaining the accuracy in interpolation, as compared to standard regression-based techniques.

Finally, we investigate estimating a component's operational profile during design by utilizing a variety of available information sources. For instance, we utilize information from domain experts, requirements document, simulation, and functionally similar components, and apply an HMM-based approach in estimating operational profiles of a component. We evaluate the effectiveness of the 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 results indicate that the framework can meaningfully assess reliability of a component 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.

1.5 Roadmap

We discuss related work in Chapter 2, and SHARP in Chapter 3. In Chapter 4, we describe our approach to performance analysis of third-party WSs. This is followed by our approach on operational profile estimation, and its application to component reliability modeling in Chapter 5. Finally, we discuss future research directions, and conclude in Chapter 6.

Chapter 2

Related Work

This chapter describes existing works in more detail, especially on design-level approaches and testing-based techniques, because of their relevance to the work proposed in this dissertation. In Chapter 2.1, we describe existing design-level software reliability analysis techniques, and highlight their solution cost and assumption on the availability of model parameters. Then, we focus on testing-based approaches and their applications to performance estimation of software in the "Binaries Accessible" category in Chapter 2.2.

2.1 Design-Level Software Reliability Analysis

As discussed in Chapter 1, it is important to start analyzing system quality early to save development cost. To this end, many design-level software reliability analysis techniques have been proposed. These approaches include [20, 33, 32, 29, 35, 36, 39, 41, 42, 28, 58, 59, 62, 66, 65, 76, 78]. A comprehensive description of these can be found in existing surveys on the topic [30, 37, 38] and the references therein. At a high level,

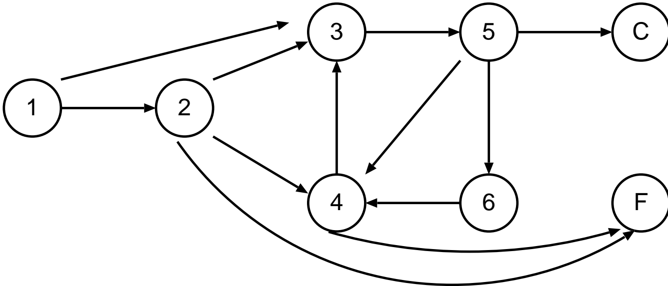


Figure 2.1: An example of the Cheung's model [20]

they make use of the system's structure in predicting system reliability. Many of them are influenced by [20], which is one of the earliest works on design-level reliability prediction that considers a system's internal structure using discrete-time Markov chains (DTMCs). An example of a model built using [20] is depicted in Figure 2.1. The states in the reliability model represent components, while the transitions represent transfer of control between components. These transitions are assumed to follow the Markov property (i.e., a transition to the next state is determined only by the current state).[1] Each component may fail with a failure probability f_i, and a transition from State i to a failure state F represents a system failure. Since failures are assumed to be irrecoverable in [20], the failure state is an absorbing state that has no outgoing transition. When the system has finished its execution, it transitions to a correct state C, which is an absorbing state that represents the system has completed without error. Therefore, system reliability is defined as the probability of eventually reaching C, which can be computed using standard techniques [69].

A number of approaches have built upon [20]. For example, in [58], instead of assuming the reliabilities of components are available, it considers each component to be providing a number of services, and computes a component's reliability by combining the reliabilities of its services. Another example is [21], which has considered error propagation in computing system reliability. They argued that an error that is caused by a component may not cause that component (and hence the system) to fail immediately. Rather, an error may propagate to other components, which then causes system failure.

[32] proposed another approach using Markov chains. Instead of computing system reliability directly from a Markov chain as in [20], they compute the number of visits to each component before the system has terminated. System reliability can then be

[1] [37, 30] have suggested that this assumption does not hold in many software systems. Higher-order Markov chains can be used to alleviate this problem, but the size of the model grows much faster, and the resulting model may be intractable.

computed by combining the components' reliabilities, accordingly to the number of visits to each component. An advantage of this approach is that when a component's reliability changes, we only need to multiply the component reliabilities, and do not need to solve the entire Markov chain again.

The approaches we have discussed thus far are classified as *state-based* approaches in [37]. [37] has classified other approaches as *path-based* approaches. In path-based models, such as [39], system reliability is defined as the weighted sum of the reliability of each execution path, and the weights represent the probabilities that each path is executed. Some path-based approaches are tied with the system's use-case scenarios. A use-case scenario describe the interactions between components to achieve a subset of the system's functionalities, and a standard way to specify scenarios is to use sequence diagrams. For example, [35] converts a sequence diagram into a DTMC, and defines system reliability as a weighted sum of scenario reliabilities. Using path-based or scenario-based approaches allow system designers to analyze the reliability of a small part of the system at a time, and may reduce the complexity in solving the model.

2.1.1 Applicability to Concurrent Systems

All approaches we have discussed so far assume a sequential system. Typically, in such approaches a reliability model keeps track of which component is running. For example, the state-based approaches, such as [20], model transfer of control between components, and assumes only one component is executing at a time: once a component has transferred control to another component, it remains idle until control is transferred back to it again. The path- or scenario-based approaches, such as [35, 39], assume only one path/scenario is being executed at a time, as they assume the probabilities that a path/scenario executes sum up to 1.

This assumption is problematic in modeling concurrent systems, in which many components may be running simultaneously. Thus, in analyzing systems with simultaneously running components, one typically needs to keep track of the status of all components. Such an approach is taken, for instance, in [28, 76], and we refer to it as a "flat model". In [28, 59, 76], a state S in the model is described using $\mathbf{C}$ variables, where $\mathbf{C}$ is the number of components in the system, i.e., $S = (S^1, S^2, \ldots, S^{\mathbf{C}})$. In [28, 76], components are modeled as black-boxes, which are either active or idle, i.e., $S^i = 0$ when Component i is idle and $S^i = 1$ when Component i is active. In addition to scalability problems, this is also a shortcoming since representing the internal structure of components facilitates more accurate models. For example, some defects may only be triggered when the component performs certain functions, and thus not having a sufficient level of granularity in the reliability model could lead to poorer reliability estimation. To address this, instead of modeling the status of a component as either active or idle, one can use a finer-granularity component model; this would result in the type of model used in [59], where S^i represents the state of Component i. Specifically, [59] generates component models from scenario sequence diagrams and then generates a system model by combining the component models using the parallel composition. We note that, as in [59], SHARP (Chapter 3) models systems at a finer granularity level through the use of scenarios. However, unlike [59], we employ a hierarchical approach, which is intended to yield better scalability. Since such approaches essentially generate reliability models in a brute-force manner, they suffer from scalability (i.e., "state explosion") problems. As a result, such models are often prohibitively costly to generate and solve, even for systems with a modest number of components.

Other state-based approaches, such as those based on stochastic Petri nets (SPNs), suffer from the same state explosion problem. Existing SPN-based approaches focus on performance analysis based on UML models (see [10] for a survey); such models can

be used in reliability analysis as well. However, solving the SPN requires generating the SPN's reachability graph, which has the same state explosion problem described above.

While non-state-based approaches, such as [39, 62, 77], may not have the state explosion problem and (implicitly) consider concurrency, they are not as descriptive as state-based approaches, and hence may not give accurate estimates. For instance, the work in [39] computes system reliability as a weighted average of the reliabilities of all execution paths, and the reliability of each path is the product of component reliabilities. In addition, [62] explored the use of Bayesian Networks (BNs) to model reliability of concurrent systems. States in the component state diagrams are interpreted as nodes in a BN, and transitions are interpreted as dependencies between nodes. The BN can then be solved for reliability given these dependencies and component reliabilities. However, the notion of concurrency in these approaches is limited as they do not describe flow of control. For example, the two approaches above are not able to model the time spent in each component, so that a lightly-used component has the same effect on reliability as a heavily-used component.

At the same time, the notion of system failure is more complex in a concurrent system. In existing approaches that assume sequential systems, failures are represented by transitions to a failure state in the (Markov-chain based) reliability model. In these models, which assume a single-threaded system, being in state S_i indicates that Component i is active, while all other components are idle. A transition from a state S_i to a failure state indicates that Component i has failed. This means that if any active component has failed, the entire system is considered to have failed. On the other hand, in a concurrent system, since more than one component may be running, assuming the system has failed if any active component has failed may be inflexible. However, existing approaches that are applicable to concurrent systems are not able to handle complex failure conditions. For example, in [59], the system transitions to a failure state when any active component

fails. The work in [28, 32] does not include failure states explicitly; rather essentially a reward is assigned to each state (with the value of the reward representing the probability of the system failing in that state), where the system's reliability is computed as a Markov reward function [69]. This system failure description is also limited, which assumes that the system fails when any (active) component fails. [76] provides a somewhat richer description of system failures, where a reliability model includes backup components that can provide services when the primary component fails; the system fails when the primary component and all backup components fail. Unfortunately, this approach is not capable (without significant changes) of describing other notions of system failure, e.g., an OR-type relationship (the system fails when Component A or Component B fails).

2.1.2 Parameter Estimation

Another common theme across these design-level reliability approaches is that it is not clear how the model parameters are estimated. This is a major challenge in design-level software quality analysis: since the implementation is not available, it is hard to gather information for analysis.

The parameters that are needed for software reliability analysis are (1) the system's operational profile, which corresponds to, for example, transition probabilities in [20], as well as probability that a scenario executes in scenario-based approaches, such as [78]; and (2) component reliabilities, which corresponds to transition probabilities to a failure state in [20].

Existing design-level reliability estimation approaches (sometimes implicitly) assume that the system's or its components' operational profiles are known. A system's operational profile is typically estimated after the system has been implemented [52]. Typically, this involves analyzing the system's traces, to determine, for example,

the transition probabilities between components. Estimating an operational profile of a system becomes non-trivial at the design level, because the implementation of a system is unavailable.

To determine a component's reliability, we can rely on component reliability prediction approaches, such as [19] (Chapter 5), to predict component reliability. However, as in the system-level, it is challenging to estimate the component's operational profile and failure information.

Some existing work acknowledge this fact, and study the effect of uncertainties about a system's operational profile on the resulting reliability estimates. For example, [36] provides an analytical evaluation of the effect of uncertainties in model parameters on the resulting system reliability. Others assume a fixed operational profile and varying component reliability, and apply traditional Markov-based sensitivity analysis [20, 66].

[60] proposes an approach to using hidden Markov models (HMMs) [55] in estimating a component's operational profile. However, it is not clear how "training data", an input to the Baum-Welch algorithm (a parameter estimation algorithm in HMMs [55]), can be obtained. Part of our contribution in this context (see Chapter 5) is to identify how training data can be obtained at the design level.

2.2 Testing-Based Software Performance Estimation Techniques

There is a vast literature on software performance evaluation, going back to [67], which proposed the software performance engineering process that has been in wide use; it examines issues in software performance evaluation, e.g., information gathering, model construction, and performance measurements. More recently, research has focused on

performance evaluation using software architectural models, e.g., [10] provides a representative survey on the topic. These works leverage software architectural models of their choice to generate performance models and focus on performance evaluation from a system designers' perspective — this allows early performance evaluation which aids in avoiding costly design problems. Given the scope of our work in Chapter 4, here we discuss works that have focused on performance evaluation of third-party WSs. Although there has been significant interest in this topic, the main shortcomings of existing techniques (as detailed below) include (a) high cost of measurements at high workloads (needed by those techniques to estimate system response time) and (b) assumptions made by those techniques about availability of information about third-party WSs.

Several black-box approaches consider predicting performance of third-party components, where the performance model is built from the component's documentation [54], or by examining the component's binary code (e.g., Java bytecodes) [40]. However, these approaches assume the availability of design models, documentation, or binaries of a third-party component, which are typically unavailable in the case of a third-party WSs. Thus, they are not readily applicable.

In [17] an approach to WS performance evaluation is proposed; however, it requires testing WSs at high workloads, which is expensive. In [68], a simulation-based approach to estimate WS response time is proposed, where results from performance testing are used when the WS being tested is lightly-loaded, to obtain simulation parameters, and predicting response time for heavier loads is done using simulation. However, a shortcoming of this work is the assumption (when generating the simulation model) of knowledge of the architecture of the WS being tested. Moreover, simulations could take a fairly long time to converge, and thus at design time, analytic techniques may be more desirable. In [75] a queueing network-based model of a composite WS is generated, in which each WS is modeled as a server in the queueing network. However, if the WS

being tested is a third-party WS, it is not clear how information about the structure of a composite WS can be gathered (e.g., to what other WSs the WS under testing makes requests).

Another approach is to include performance information in a WS's service description, so that their clients can use such information for performance evaluation. For example, [24] proposes that P-WSDL includes service performance characteristics of the system (e.g., utilization and/or throughput), network information (e.g., network bandwidth), and workload characteristics (e.g., request arrival rate). We argue that service providers may be reluctant to provide such information, and it is not clear how this information can describe a composite WS, in which the service performance depends on other WSs. Lastly, [46] proposes to include demands on server resources for each interfaces (e.g., a service requires X units of CPU time and Y units of I/O). Unfortunately, it is not clear how the service demand can be obtained, as it is difficult to map a high-level service to low-level hardware demands.

Existing work on performance evaluation specific to Web applications focuses on evaluation from a service provider's perspective. For example, [74] models a Web application as a multi-tier system. Each tier represents a different type of server, and the routing between the servers are assumed to be known. In such efforts, the goal of the analysis is to help service providers to make service-level agreements (SLA) for their clients. Our approach is orthogonal in that we evaluate a WS from the client's perspective.

The work in [56] monitors the quality of a third-party WS, including performance, to detect violations of a SLA. This work is similar to ours in the collection of statistics to evaluate a third-party WS. On the other hand, they assume the system is already operational, and are interested in the distribution of performance measures (e.g., response

time distribution) to look for SLA violations, while we use the information to predict the performance of a WS to aid service selection.

Chapter 3

SHARP: A Scalable Framework for Reliability Prediction of Concurrent Systems

We present SHARP, a scalable, hierarchical, architecture-level reliability prediction framework for concurrent systems. We first present background information on a running example we use throughout this chapter, the MIDAS system [45], along with information on software design models and architecture modeling in Chapter 3.1. This is followed by an overview of the SHARP framework in Chapter 3.2, and the details are described in Chapter 3.3. Finally, we evaluate SHARP in Chapter 3.4.

3.1 Background

For the ease of exposition of our framework and for its subsequent evaluation, we use a sensor network application, built using the MIDAS framework [45] and depicted in Figure 3.1, as our running example. This system monitors room temperature and turns the air conditioner (AC) on or off in order to satisfy user-specified temperature levels.

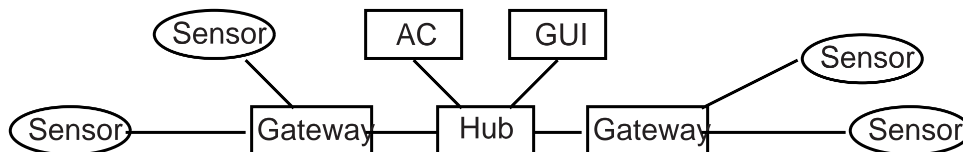
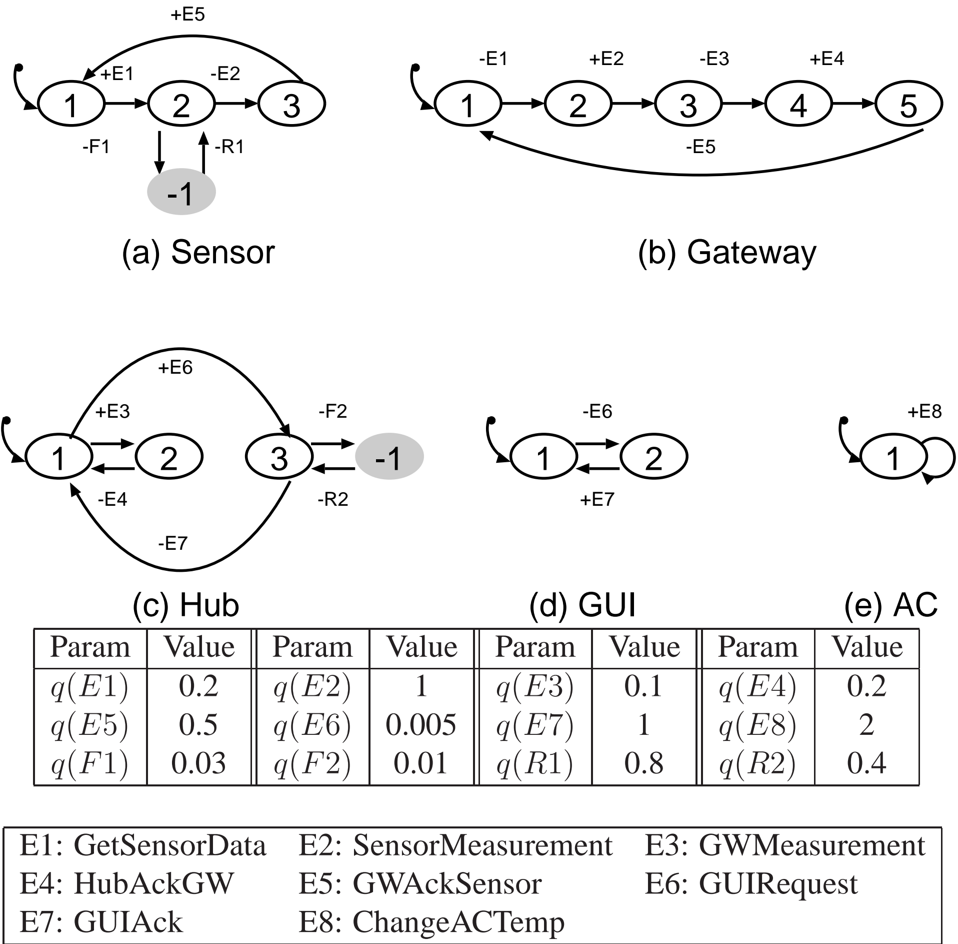


Figure 3.1: An Overview of the MIDAS system



Param	Value	Param	Value	Param	Value	Param	Value
$q(E1)$	0.2	$q(E2)$	1	$q(E3)$	0.1	$q(E4)$	0.2
$q(E5)$	0.5	$q(E6)$	0.005	$q(E7)$	1	$q(E8)$	2
$q(F1)$	0.03	$q(F2)$	0.01	$q(R1)$	0.8	$q(R2)$	0.4

E1: GetSensorData	E2: SensorMeasurement	E3: GWMeasurement
E4: HubAckGW	E5: GWAckSensor	E6: GUIRequest
E7: GUIAck	E8: ChangeACTemp	

Figure 3.2: Components' state diagrams

We refer to this example system as the MIDAS system. The MIDAS system consists of five different types of components: a *Sensor* measures temperature and sends the measured data to a *Gateway*. The *Gateway* aggregates and translates the data and sends it to a *Hub*, which determines whether it should turn the *AC* on or off. Users can view the current temperature and change the thresholds using a *GUI* component, which then sends an update to the *Hub*.

The state diagrams capturing the behavior of the MIDAS components are given in Figure 3.2. In a component state diagram, an event E is either a sending event or a

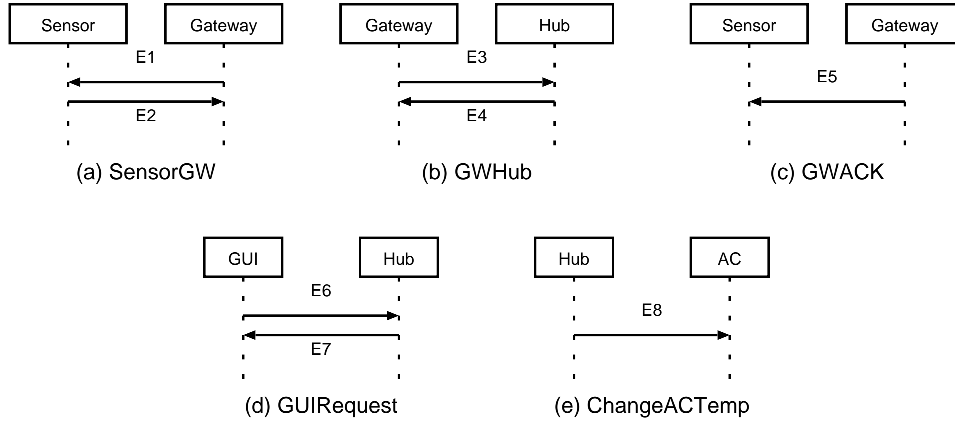


Figure 3.3: Sequence diagrams

receiving event. In this paper, we use the notation introduced in [79], in which sending and receiving events is represented by "-" and "+", respectively. In SHARP, an event needs to come with a specification of its arrival rate in states in which that event is enabled. These rates should be available from different information sources that are available during a system's design; further assessment and discussion of these information sources is detailed in Chapter 5. Some of the state machines in Figure 3.2 include failure states (labeled by a negative number) that represent erroneous behavior triggered by a failure event F. In the following section, we discuss how we derive the failure states.

The system-level behavior of MIDAS is captured using five basic scenarios: the *SensorGW* scenario that includes processing measurements from a *Sensor* component by a *Gateway* (Figure 3.3(a)); the *GWHub* scenario that includes processing aggregated measurements from a *Gateway* to the *Hub* (Figure 3.3(b)); the *GWACK* scenario that includes acknowledging the *Sensor*'s measurement (Figure 3.3(c)); the *GUIRequest* scenario that includes updating the temperature readings and changing temperature thresholds (Figure 3.3(d)); and the *ChangeACTemp* scenario that includes turning *AC* on or off according to the temperature readings (Figure 3.3(e)).

The five basic scenarios are in turn combined to form more complex system behaviors as shown in Figure 3.4. The complex system behavior consists of relations between basic and complex scenarios that altogether form a scenario hierarchy [1]. The different scenarios can run concurrently (PAR relationship) or sequentially one after the other (SEQ relationship). In MIDAS, complex scenario *Sensors_PAR* represents the parallel execution of multiple *Sensors* running the *SensorGW* scenario (Figure 3.4 describes a system variant with four sensors). *Sensor_PAR* is considered complete once all the concurrently running scenario instances are complete. Furthermore, the complex scenario *SensorMeasurement* specifies a longer sequence that summarizes how a sensor measurement is propagated from *Sensors* to *Gateway* to *Hub* and back.

Hierarchical scenario descriptions similar to the one presented for MIDAS are commonly created during system design. In SHARP, an engineer also needs to annotate the scenario hierarchy with the following quantitative information: (1) branching probabilities when one scenario can be sequentially followed by multiple other scenarios, and (2) the number of scenario instances that run in parallel. While other approaches require and utilize the branching probabilities (e.g., [59]), SHARP is unique in its ability to effectively handle scenario multiplicities. The information about how many scenarios of a certain type will be running in parallel should be derivable either from the system requirements or from the architectural configuration.

[1] We assume that the system designers have already handled implied scenarios [73] and updated the specification to be free of them.

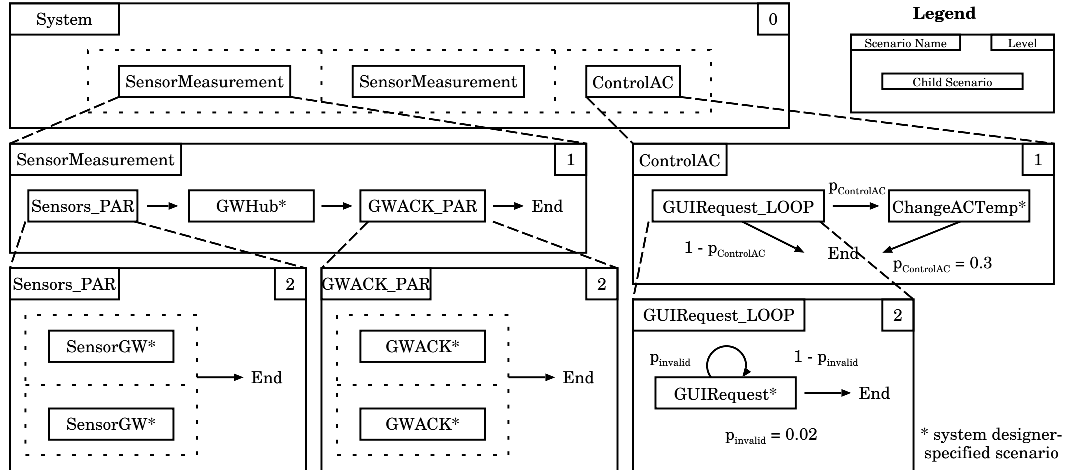


Figure 3.4: MIDAS scenarios organized in a hierarchy

3.1.1 Architectural-Level Defect Analysis

In this paper, we focus on analyzing the reliability effect of *architectural defects* [63], such as operation signature mismatches and mismatches between components' interaction protocols. We define *system reliability* as the probability that a user does not experience a failure caused by architectural defects. When a defect is triggered, a *failure* may occur, after which a component behaves erroneously with respect to the system's requirements. SHARP accounts for the fact that a *recovery* from failure is possible.

In system reliability analysis, a system's failure is typically defined in terms of the failures of its components. We analyze an individual system component's architectural model by applying a defect classification technique [63] to determine the failure states. We then add a failure transition from all the states in which a defect may be triggered. For example, using the defect classification technique [63] on MIDAS, we determine that a *Sensor* is unable to notify the *Gateway* when it is running out of battery. This defect was discovered as a mismatch between the two components' interaction protocols. Failures caused by this defect are represented as the failure state -1 in Figure 3.2(a). Furthermore, *Sensor* returns to State 2 upon recovery.

In general, a component can return to any state designated by the system designer during defect analysis. We extend the event send/receive nomenclature to failure and recovery events by viewing a component as sending failure (recovery) events when it fails (recovers). We assume that the failures are recoverable, but can model irrecoverable failures without significantly changing SHARP; only a few equations in Chapter 3.2 would need to be updated, where transient [72] (rather than steady state) analysis would be used. For brevity, in this paper we omit details of irrecoverable failure analysis.

3.2 An Overview of the SHARP framework

The SHARP framework estimates a system's reliability based on (a) a behavioral specification provided primarily as a scenario hierarchy, (b) an operational profile, and (c) a definition of failure states. At a high level, SHARP works by partitioning the system behavior into smaller analyzable parts according to the scenario specification, with the premise that analyzing multiple smaller models is more efficient than analyzing one very large model. This is in stark contrast to a state-of-the-art scenario-based technique [59] that generates a full-blown reliability model from a complex scenario specification. Conversely, the idea of state space partitioning using scenarios has been explored in the literature [23, 78] but with notable shortcomings. Specifically, the existing research does not resolve two crucial obstacles to reliability estimation of complex, concurrent systems:

1. How can one efficiently solve a reliability model that captures complex system behaviors consisting of elaborate multi-scenario sequences when solving the whole model at once does not scale?

2. How can one efficiently estimate system reliability when a system consists of tens or even hundreds of concurrently running components and scenarios with possibly similar behavior?

The first obstacle corresponds to the ability to deal with sequential scenario combinations without having to solve the corresponding "flat" model used by other approaches [59] and without making simplifying assumptions about scenario independence like some existing techniques [23]. The second obstacle relates to the need to handle situations in which multiple scenario instances are running concurrently (PAR relationship). For example, we want to be able to efficiently solve *Sensors_PAR* scenario from Figure 3.4 even in situations when we have thousands of concurrently running *Sensors*. SHARP resolves both of these obstacles by first generating and solving the reliability models of smaller scenarios and then incorporating the results into reliability models of the complex scenarios; this is done in a bottom-up way throughout the specified scenario hierarchy.

To solve for reliability of a complex scenario with sequential dependencies (e.g., the *SensorMeasurement* scenario) in which there may be a large number of scenarios running one after another, we propose a technique based on stochastic complementation [49]. Stochastic complementation is a standard technique for solving large Markov chains that relies on partitioning a large model to smaller analyzable parts that have a low number of incoming and/or outgoing transitions in the original model. To be able to do this, we utilize the partitioning that is intrinsically present in a SEQ scenario where each sub-scenario has only one entry point. For example, when analyzing MIDAS scenario hierarchy (Figure 3.4) SHARP utilizes the SEQ relations in the *SensorMeasurement* scenario to solve *Sensors_PAR*, *GWHub*, and *GWACK_PAR* first, and then incorporate the obtained results into a small, high-level *SensorMeasurement* model with only three

states. The outlined method for estimating reliability of complex SEQ scenarios demonstrates the synergy between structured software specifications and stochastic methods that is present in our framework.

To estimate the reliability of a PAR scenario (e.g., *Sensor_PAR* in Figure 3.4), instead of generating parallel scenario models by simply composing all of the instances together [43] or keeping track of the internal states of all components, SHARP works by keeping track of the number of concurrently running scenario instances. To be able to abstract a scenario's execution state to either running or completed, SHARP calculates completion rates of the different scenarios using queueing network (QN) models [69]. For example, we aggregate the overall behavior of *Sensors_PAR* from Figure 3.4 with a model that tracks whether there are zero, one, or two concurrently running instances of *SensorGW*. The transition rates between these different *Sensors_PAR* states amount to the previously calculated *SensorGW* completion rate. In certain cases (e.g., very large scale systems), even the described symbolic model can become intractably large. SHARP applies model truncation [69] on such models. Model truncation removes the rarely visited states (i.e., rare scenario combinations) thus achieving notable scalability gains with a minimal loss in accuracy.

Moreover, previous techniques fail short when analyzing concurrent systems because they do not consider that components in such systems often compete for the same set of resources. To address this, we take resource contention into account. Specifically, SHARP includes information about the possible concurrently running scenarios when constructing the basic scenario QNs. For MIDAS, we identify that the *Gateway* is a possible contention point as multiple *Sensors* may be using it concurrently; SHARP includes this information when building the *SensorGW* QN and reliability model.

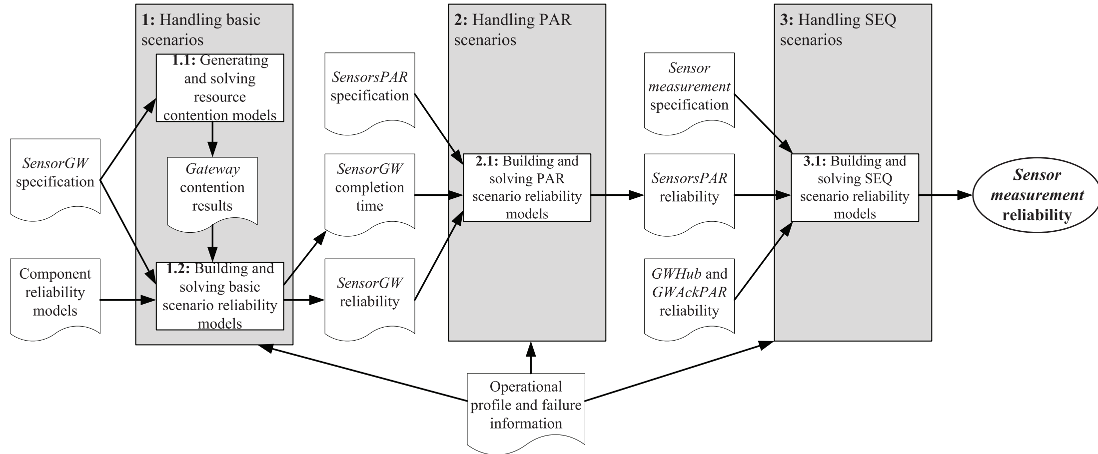


Figure 3.5: An illustration of SHARP applied on the complex *Sensor measurement* scenario

3.3 Reliability Computation

In this section, we present the technical details of SHARP. SHARP has three distinct activities that target (1) basic scenarios, (2) SEQ complex scenarios, and (3) PAR complex scenarios. We describe each activity in Chapters 3.3.1, 3.3.2, and 3.3.3, respectively. Each of these activities consists of steps for solving the corresponding models for reliability and completion time; the completion times are used when solving the PAR scenarios as elaborated below.

As an example, Figure 3.5 illustrates the steps that SHARP performs to analyze the reliability of the *Sensor measurement* scenario from Figure 3.4. The different SHARP activities are used to analyze the different parts of the scenario hierarchy. The process for obtaining the reliability information for *GWHub* and *GWACK_PAR*, for brevity not shown in Figure 3.5, is identical to the process for *SensorGW* and *SensorsPAR*. Intuitively, SHARP first analyzes the low-level, basic scenarios and incrementally incorporates the lower-level analysis results in the higher-level SEQ and PAR scenario models. Note that each activity comprises efficient steps for analyzing the scenario reliabilities and completion times, while the basic scenario activity also contains a contention

modeling step. In the following sections, we describe the SHARP activities with their corresponding steps. The order of applying the different activities ultimately depends on the structure of the scenario hierarchy.

3.3.1 Basic Scenarios

As discussed earlier, the first step in solving for system reliability in SHARP is to solve for the reliability and completion time of the basic scenarios. We generate the scenario-based reliability models (SBM for short) in a similar manner to existing research [23, 59, 78] (described as Step 1.1 in the following section). The unique aspect of our approach is that we generate the failure states for a basic scenario based on the correspondence between the component reliability models (e.g., Figure 3.2) and the generated SBM. By doing so, we manage to reuse component-level information about architectural defects thus making the reliability analysis more meaningful as opposed to having an engineer "guess" the failure states.

Next, we augment the generated SBM to model *resource contention* with special "queueing" states (Step 1.2). Intuitively, such states simulate a situation when an event cannot be processed immediately. For example, while the *Gateway* is processing data from one *Sensor*, it may receive data from another; consequently, we augment the SBM by adding a "queueing" state to represent queueing of the *Sensor*'s request. Contention-related parameters are computed using queueing networks (QNs) [69]. For example, to compute the average waiting time of a *Sensor* request, we build a QN model depicted in Figure 3.8. Parameters needed to build this model, e.g., frequency and the processing time of requests, are derivable from the operation profile. This contention-related behavior is included with minor increasing reliability model's size as the QNs are solved separately and only their results are "plugged" back into the reliability model.

(a) Sensor (b) Gateway

Figure 3.6: Component submodels of *SensorGW*

Once the SBM reliability model is constructed and the QN is solved for contention, we solve SBM for scenario reliability (Step 1.3) and completion time (Step 1.4) using standard methods.

Step 1.1: Generating SBM

To generate the SBM for a scenario, we first generate a component submodel for each component in each scenario, and then apply parallel composition, as in [59]. A component submodel of Component $Comp_c$ in Scenario $Scen_i$, $Comp_c_Scen_i$, is a state machine model describing the behavior of $Comp_c$ in $Scen_i$, in which a state transition represents the occurrence of an event in the corresponding sequence diagram. In our MIDAS example, the component submodels for the *SensorGW* scenario (recall Figure 3.3) are depicted in Figure 3.6.

The next step is to add failure states to each component submodel. We identify the possible points of failure in a component submodel by leveraging the component model using our work [19]. As we discussed in Chapter 3.1, we assume we have identified the architectural defects in the components, and represent them as failure states in a component model. For example, to model the defect in the *Sensor*, represented by the failure state -1 in Figure 3.2(a) (recall Chapter 3.1), we add a failure state, State -1, in Figure 3.6, a failure transition from State 2 to State -1, and a recovery transition from State -1 to State 2.

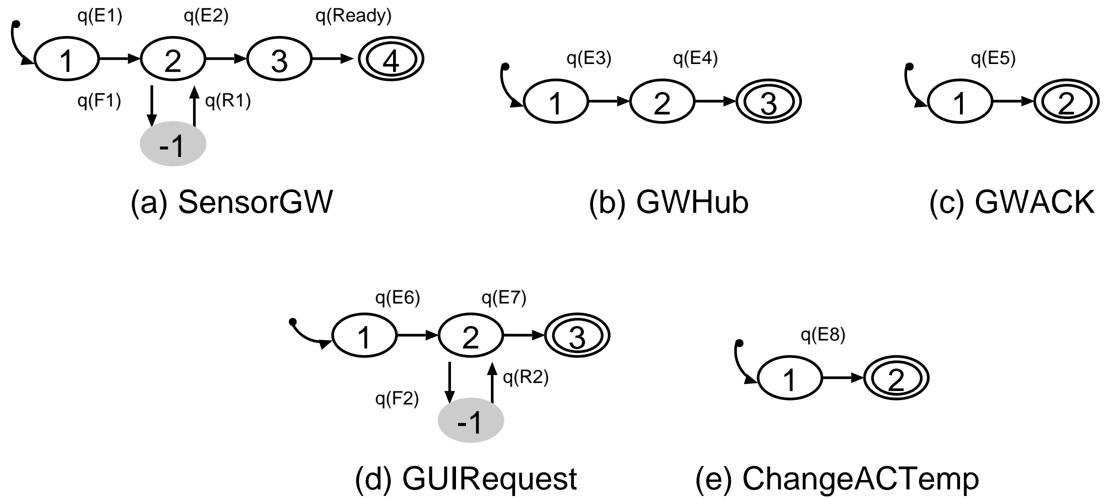


Figure 3.7: SBMs of the basic scenarios

We then generate an SBM for each $Scen_i$ by applying parallel composition [43] to the component submodels $Comp_c_Scen_i$ for all $Comp_c$. In MIDAS, the SBM of the five scenarios specified by the system designer is depicted in Figure 3.7. In our example, applying parallel composition to the component submodels in Figure 3.6 would result in the SBM for the *SensorGW* scenario depicted in Figure 3.7(a). Note that the states corresponding to normal behavior are marked in white, while the failure states are marked in grey.[2] (Note that State 3 in the *SensorGW* scenario (Figure 3.7(a)) corresponds to contention modeling, which we describe next).

A major difference between our approach and [59] is that [59] first combines the component submodels of each component for all scenarios (i.e., combining the models $Comp_c_Scen_i$ for all $Scen_i$ to synthesize a model for $Comp_c$), and then apply parallel composition to all synthesized component models. Conversely, we apply parallel composition to the component submodels for each scenario separately (i.e., for each $Scen_i$, we apply the algorithm to $Comp_c_Scen_i$ for all $Comp_c$); we then combine

[2]For ease of presentation, we assume that the execution of a scenario has failed if any component is in a failure state. SHARP is flexible enough to allow designers to specify more complex failure rules. This is done using the same technique in PAR scenario as described in Chapter 3.3.3.

the results of solving the SBMs by solving its parent's SBM. As a consequence, our approach addresses the common scalability problems because generating and solving many, smaller models, rather than a huge model in [59], results in space and computationally savings, as discussed earlier.

Finally, we determine the transition rate between the states based on the provided operational profile. Formally, let Q_i be the transition rate matrix for $Scen_i$'s SBM. If the transition from State j to State k corresponds to the event E, the transition rate $Q_i(j, k) = q(E)$, where $q(E)$ is the rate that event E occurs. To complete the SBM, according to [69], we set the diagonal entries $Q(j, j)$ such that each row in Q sums to zeros.[3]

Step 1.2: Modeling Contention

To model contention in SHARP, we augment the SBMs with contention information. When several components (callers) request services from a servicing component (callee), the callee needs to allocate its resources appropriately to serve a caller, while other callers would need to wait to obtain service.[4] Since the system behavior may be different when a request is waiting for service and when it is being processed, we add a *queueing* state to represent that a caller's request is queued. Formally, let E be an event that triggers a transition from State j to State k in an SBM. If there is a component that may be servicing other requests upon receiving E, we add a queueing state State q such that $Q(j, \mathsf{q}) = q(E)$, and $Q(\mathsf{q}, k) = q(Ready)$. $Ready$ is an event indicating that the callee is ready to process the request of the caller of interest. As an example, in

[3] Note that self-loops in a component model (i.e., an event that causes a transition from a state to itself) have been implicitly accounted for here. Since self-loops do not cause any state transitions, they do not affect the probability distribution of being in a state in a CTMC, and are therefore dropped in a SBM.

[4] Since the flat model results in callees serving the callers on a FCFS basis, we also use FCFS in our exposition (as the flat model is used as the ground truth); however, SHARP allows other queueing disciplines, which can be modeled similarly.

the *SensorGW*, the *Gateway* could be servicing a *Sensor*'s request when another *Sensor* triggers event $E1$. Therefore, we add State 3 to represent queueing, as in Figure 3.7(a). Any other points of contention would be modeled in a similar manner.

The next step is to determine $q(Ready)$, the outgoing rate of the queueing state. We define $q(Ready) = \frac{1}{T_{wait}}$, where T_{wait} is the average time a caller spends waiting to receive service. To compute T_{wait}, we solve a *queueing network* (QN) [69], which describes the queueing behavior of the callers' requests. In this QN, the callee is represented by a server.

To build such a QN, we utilize the following information: (a) the number of different types of callers (i.e., the different types of components where each type can request different services with different processing times), and the maximum number of type of caller that may request a service; (b) how often a caller requests a service (arrival rate); (c) how long the callee takes to serve a request (service rate); and (d) the callee's *queueing discipline*. Note that (a) is available from the system's requirements and architectural models which contain architectural configuration information, (b) is the available from the operational profile (i.e., the rate of an event E), and (c) is the total rate leaving a state k also derivable from the operational profile. The operational profile information and the other model parameters are determined using an approach to be described in Chapter 5. Lastly, (d) describes how the callee's resources are allocated among callers. For example, the callee can serve the incoming requests in a first-come-first-serve (FCFS) or a round-robin (RR) fashion. This information should be available from the system's requirements and the architectural models. For example, the choice of a given middleware for implementing the system may impose FCFS service of callers. The constructed QN is finally solved for the average waiting time in the queue using standard methods [69].

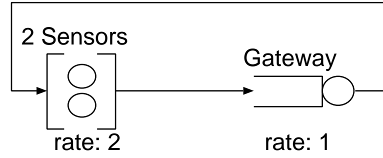


Figure 3.8: QN model of the *SensorGW* scenario

The QN for *SensorGW* basic scenario is depicted in Figure 3.8. As an example, we are interested in modeling the case when a *Sensor* sends measurements to the *Gateway* while the *Gateway* is processing another *Sensor*'s request. Hence, we have one class of callers: two *Sensors* may send measurements to the *Gateway* with arrival rate being $2 \times q(E2) = 2$, processing rate being $q(E3) = 1$, and the *Gateway* is a FCFS callee. After solving the QN for the average waiting time at *Gateway*'s queue in Figure 3.8, the resulting rate of leaving the queueing state (State 3) in Figure 3.7(a) is estimated to be 5. Other points of contention in the SBMs are treated analogously.

Step 1.3: Computing Scenario Reliability

Scenario reliability is defined as the probability of not being in a failure state. Solving for it involves finding the steady state solution of a SBM. Note that the process we apply to compute basic scenario reliability needs to take into account that, at a higher level, we utilize stochastic complementation [49] to handle SEQ scenarios. Intuitively, stochastic complementation breaks a large Markov model into a number of submodels, solves the submodels separately, and reconstructs results of the original model. The special structure required for an efficient solution using stochastic complementation is that each submodel has only one start state. Notably, the generated basic scenario SBMs satisfy this requirement as they have a single starting state.

To obtain the basic scenario reliability, we assume a system is executing for a long time (i.e., it is in its steady state), that after the execution of $Scen_i$ we are analyzing,

it will eventually executes $Scen_i$ again. Therefore, at the level of a basic scenario, we are interested in finding its reliability r_i, which is the conditional probability of not being in a failure state, given that $Scen_i$ is executing. To complete the model, we need to account for the transition going out of $Scen_i$ and determine the state to which the control is eventually transferred from another scenario. In case of a basic scenario, the execution always returns to the scenario's Start state. The execution of a scenario ends when it is about to go to an End state, which represents the start of another scenario. Therefore, based on [49], we remove the End state, and redistribute the transition rates to the Start state (i.e., $Q_i(j, End) = Q_i(j, 1)$). For example, in *GUIRequest*'s SBM in Figure 3.7(d), we replace the transition from State 2 to State 3 with a transition from State 2 to State 1, with the transition rate being $q(E7)$, as in Figure 3.9. Now we can solve this model for its steady state probability vector, $\vec{\pi_i}$), by using standard techniques [69], and scenario reliability r_i can be computed as follows:

$$r_i = 1 - \sum_{f \in F_i} \pi_i(f) \tag{3.1}$$

where F_i is a set of failure states in $Scen_i$'s SBM. The rate matrix of *GUIRequest*'s model in Figure 3.9 after rate redistribution is

$$\begin{matrix} 1 \\ 2 \\ -1 \end{matrix} \begin{bmatrix} -0.005 & 0.005 & 0 \\ 0.1 & -0.11 & 0.01 \\ 0 & 0.4 & -0.4 \end{bmatrix} \tag{3.2}$$

Solving this matrix using standard technique for $\vec{\pi_i}$ would yield $\vec{\pi_i} = [0.9512, 0.0476, 0.0012]$, and hence the reliability of the *GUIRequest* scenario is $r_i = 1 - 0.0012 = 0.9988$.

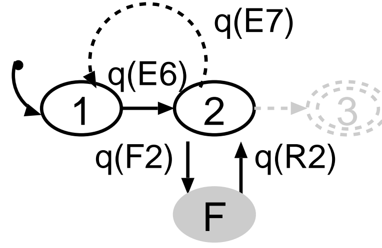


Figure 3.9: Rate redistribution in *GUIRequest*

Step 1.4: Computing Scenario Completion Time

Recall that the scenario's completion time, t_i, is needed in building the concurrency-level models for a PAR scenario (Chapter 3.3.3). We can compute t_i from $Scen_j$'s SBM, after updating its parameters. We note that t_i includes time spent in normal operation as well as time spent in recovering from failures, because the definition of the "completion" of a scenario includes all of the scenario's behavior.

Let $T_i(s)$ be the completion time when we are in State i in $Scen_i$'s SBM. i.e., $t_i = T_i(1)$. We compute $\vec{T_i}$ by performing transient analysis that corresponds to solving Eq. (3.3) [72]:

$$Q_i'\vec{T_i} = -e \tag{3.3}$$

where Q_i' is the matrix after eliminating the row and column corresponds to the End state in Q_i, and $-e$ is a column vector of -1 with the appropriate dimension. Consider the *GUIRequest* scenario, the rate matrix Q_i' is

$$Q_i' = \begin{matrix} 1 \\ 2 \\ -1 \end{matrix} \begin{bmatrix} -0.005 & 0.005 & 0 \\ 0 & -0.11 & 0.01 \\ 0 & 0.4 & -0.4 \end{bmatrix} \tag{3.4}$$

Table 3.1: r_i and t_i of the MIDAS scenarios

Scenario	r_i	t_i	Scenario	r_i	t_i
SensorGW	0.9900	6.2625	GWHub	1	15
GWACK	1	2	Sensors_PAR	0.9867	9.3937
GWACK_PAR	1	3	SensorMeasurement	0.9867	27.394
GUIRequest	0.9999	201.03	ChangeACTemp	1	0.5
GUI_LOOP	0.9999	205.13	ControlAC	0.9999	205.28
System	0.9940	287.46			

Applying Eq. (3.3) to Q_i', ***GUIRequest***'s $\vec{T_i}$ is $\vec{T_i} = [201.03, 1.025, 3.525]$, and hence $t_i = 201.03$. t_i of the MIDAS scenarios are depicted in Table 3.1.

3.3.2 SEQ Scenarios

To analyze a complex scenario with sequential dependencies, we apply stochastic complementation to generate a SEQ scenario's SBM by combining the SBMs of the child scenarios (Step 2.1). This is a novel use of an advanced stochastic method for analyzing a software system's quality, and comprises an important contribution of this paper. We solve the resulting SEQ scenario SBM for scenario reliability (Step 2.2) and completion time (Step 2.3) in a similar manner to our solution for basic scenarios.

Step 2.1: Generating SBM

We generate an SBM for a SEQ scenario as follows: we first generate the states of the model, and then compute the transition rates with respect to the applied stochastic complementation [49]. The states in a SEQ scenario's SBM correspond to the child scenarios. We determine the transitions according to the dependencies between the child scenarios. If a SEQ scenario $Scen_i$ has a child scenario $Scen_k$ executing after another

child scenario $Scen_j$, we add a transition from state j to state k in $Scen_i$'s SBM. For example, the SBMs of the SEQ scenarios in MIDAS are depicted in Figure 3.10.[5]

The transition rates for each transition determined above are calculated as follows [49]:

$$Q_i(j,k) = (p_i(j,k))out_j \tag{3.5}$$

where $p_i(j,k)$ is the probability that $Scen_k$ executes after the execution of $Scen_j$, and out_j is defined in a similar manner to [49]:

$$out_j = \sum_{s \in S_j} (\pi_j(s))Q_j(s, End) \tag{3.6}$$

where S_j is a set of states in $Scen_j$, $\pi_j(s)$ is the steady state probability of being in State s in the $Scen_j$'s model, and $Q_j(s, End)$ is the transition rate going from State s to the End state in $Scen_j$'s model. For example, in ***GUI_LOOP*** (Figure 3.10(b)), let $Scen_i = GUI_LOOP$ and $Scen_j = GUIRequest$, the transition rate from State 1 to State 2 in $Scen_i$ is

$$\begin{aligned} Q_i(1,2) &= p_i(1,2)(\sum_{s \in S_j} \pi_j(s)Q_j(s, End)) \\ &= (1 - p_{invalid})(\pi_j(2))(q(E7)) \\ &= (0.98)(0.005)(1) = 0.0049 \end{aligned}$$

Furthermore, when we move up a level in the hierarchy to *ControlAC* (Figure 3.4) the transition rate going from State 1 to 2 in *ControlAC*'s SBM in (Figure 3.10(c)) becomes

[5]Note that the self-loop in State 1 of the *GUI_LOOP* scenario (depicted as dotted arrow in Figure 3.10), representing that the user's input is invalid, has been dropped, because a CTMC implicitly accounts for self-loops.

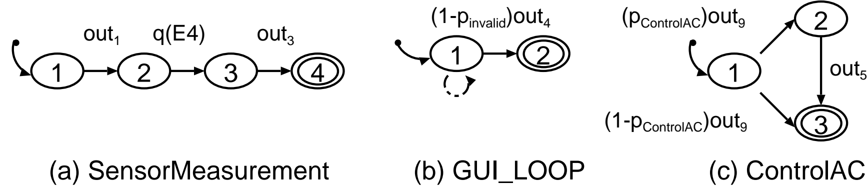


Figure 3.10: SBMs of the SEQ scenarios

$$
\begin{aligned}
Q_i(1,2) &= p_i(1,2)\sum_{s\in S_j}\pi_j(s)Q_j(s,End) \\
&= (p_{ControlAC})(1)Q_j(1,End) \\
&= (0.3)(1)(0.0049) = 0.0015
\end{aligned}
$$

Step 2.2: Computing Scenario Reliability

Similarly to the way SHARP solves the basic scenario SBM, we redistribute the rate going to the End state of a SEQ scenario SBM and solve the model for its steady state probability vector, $\vec{\pi}_i$, using standard technique. Once we have computed $\vec{\pi}_i$ and r_j for all child scenarios $Scen_j$, we solve the scenario reliability using the equation

$$
r_i = 1 - \sum_j \pi_i(j) r_j \tag{3.7}
$$

Continuing with our example, to solve for $\vec{\pi}_i$ using the SBM of the *ControlAC* scenario, after redistributing the rate going to the End state, we have the following rate matrix:

$$
\begin{bmatrix} -0.0015 & 0.0015 \\ 2 & -2 \end{bmatrix} \tag{3.8}
$$

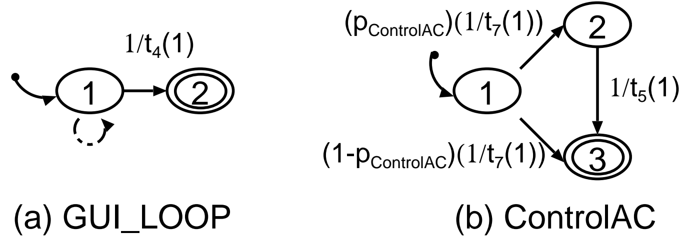


Figure 3.11: Models for completion rate computation for *GUI_LOOP* and *ControlAC*

Solving this model gives us $\vec{\pi}_i = [0.9993, 0.0007]$. The reliability of the *Change-ACTemp* is 1, as there is no defect identified in that scenario. Hence, the reliability of the *ControlAC* scenario is $r_i = (0.9993)(0.9999) + (0.0007)(1) = 0.9999$. The reliabilities of other scenarios, depicted in Table 3.1, are similarly computed.

Step 2.3: Computing Scenario Completion Time

To compute the SEQ scenario completion time, we combine the completion time of the child scenarios, represented by a state in this SBM. Formally, if State j represents a child scenario $Scen_j$, we update the rate going to another State k to $Q_i(j,k) = (p_i(j,k))(1/t_j)$, where t_j is the completion time of a child scenario $Scen_j$, and $p_i(j,k)$ is the probability of going from State j from State k in the parent scenario $Scen_i$. The difference compared to the model generated in Step 2.1 is that the transition rates here are a function of child scenario completion times, whereas in Step 2.1 the transition rates were a function of the transition rates going into the End states of the child scenarios. The reason for this change is the purpose of the analysis —reliability or completion time— which requires different information incorporated from the child scenarios. Subsequently, we compute the completion time using the standard methods (Eq. 3.3).

For example, consider the *GUI_LOOP* scenario from Figure 3.4. The rate matrix for completion time computation in Figure 3.11(a) by updating the rates in Figure 3.10(b) as follows: let $Scen_i = GUI_LOOP$, and $Scen_j = GUIRequest$, using the rate matrix

and steady state probability vector in Chapter 3.3.1, we can update the transition rate from State 1 to State 2 to $Q_i(1,2) = (p_i(1,2))(1/t_j) = (1-0.02)(1/201.03) = 0.0049$. Applying Eq. (3.3) gives us the completion time of *GUI_LOOP* is 205.12 time units.

Let us move up one level in the hierarchy and consider the *ControlAC* scenario, in which the model for completion time computation is depicted in Figure 3.11(b). Let $Scen_i = ControlAC$, and $Scen_j = GUI_LOOP$, the transition rate from State 1 to State 2 in *ControlAC* is $Q_i(1,2) = (p_{1,2})(1/t_j) = (p_{ControlAC})(1/t_j) = (0.3)(1/205.12) = 0.0015$. Similarly, the transition rate from State 1 to State 3 is $Q_i(1,3) = (p_{1,3})(1/t_j) = (1 - p_{ControlAC})(1/t_j) = (0.7)(1/205.12) = 0.0034$.

The transition rate from State 2 to State 3 is the completion rate of *ChangeACTemp*, which is $Q_i(2,3) = 1/0.5 = 2$. Given these parameters, the rate matrix of the model in Figure 3.11, Q' (defined in Chapter 3.3.1) is

$$Q' = \begin{matrix} 1 \\ 2 \end{matrix} \begin{bmatrix} -0.0047 & 0.0015 \\ 0 & -2 \end{bmatrix}$$

Thus, the completion rate of *ControlAC* after solving Q' using Eq. (3.3) is $t_i = 205.28$ time units. Note that the completion time of the *ControlAC* scenario is similar to that of the *GUI_LOOP* scenario. This is because the completion time of the *Change-ACTemp* scenario is low as compared to the completion time of the *GUI_LOOP* scenario (0.5 vs. 205.12 time units). Therefore, the *ChangeACTemp* has little impact on the completion time of the *ControlAC* scenario.

3.3.3 PAR Scenarios

The primary goal in the generation of a PAR scenario's SBM is to avoid scalability problems that arise when handling systems in which there are many concurrent scenario

instances, some of which are of the same type. To this end, we propose a method based on symbolic representation of the system execution state. Specifically, we abstract the execution of concurrent scenarios by creating a model that keeps track of currently running instances of each scenario. Each state of a PAR scenario SBM can be described as a *combination of child scenarios*, or simply, a combination. Our model also allows us to avoid redundancy in the models we generate, when the system can have several instances of the same scenario. The generated symbolic model is referred to as the concurrency-level model in the following sections. SHARP first determines the feasible scenario combinations (Step 3.1), and constructs the concurrency-level model (Step 3.2). Next, SHARP calculates the probabilities (Step 3.4) and the reliabilities (Step 3.5) of the different scenario combinations. SHARP ultimately uses the obtained information to compute the overall PAR scenario reliability (Step 3.6) and completion time (Step 3.7).

To further address the potential scalability problems when dealing with very large-scale systems for which concurrent scenario instances may be in the range of hundreds or thousands, SHARP employs model truncation [69] (Step 3.3).

Step 3.1: Determining Scenario Combinations

Determining the possible combinations is the first step in solving for reliability and completion time of a PAR scenario; as a reminder we consider a PAR scenario complete when all of its child scenarios end their execution. A combination, C_i, is defined as $C_i = (c_1, c_2, \ldots, c_{\mathbf{S_c}})$, where c_j is the number of completed instances of $Scen_j$, and $\mathbf{S_c}$ is the number of child scenarios.[6] We also define I_j to be the number of instances of

[6]Given that the system essentially experiences scenario "completions", we assume that the probability that more than one scenario completes in the exact same instant in time is negligible. This is a standard assumption in Markov chain models which makes them more tractable without a significant loss in what is expressable with such models.

Table 3.2: Values of $P(C_k)$ and $R(C_k)$ in the *System* scenario

C_k	$P(C_k)$	$R(C_k)$	t_i	C_k	$P(C_k)$	$R(C_k)$	t_i
(0,0)	0.0420	0.9606	287.46	(1,0)	0.0630	0.9735	260.07
(2,0)	0.1260	0.9866	232.67	(3,0)	0.7266	0.9999	205.28
(0,1)	0.0077	0.9607	82.181	(1,1)	0.0116	0.9736	54.787
(2,1)	0.0231	0.9867	27.394				

$Scen_j$ that needs to be completed, and $\mathbf{I} = max(I_j)$ be the largest number of possible instances among all $Scen_j$. The execution of a PAR scenario is completed only when all child scenarios has been completed.

In order to find scenario reliability, we need to compute the distribution of the possible combinations. Since, in general, not all combinations of scenarios in a system may be possible, we allow a system architect to specify the combinations of scenarios that are not possible (or not allowed). For instance, in MIDAS, such a restriction may be put in place to avoid exhausting the resources of the *Hub*: by allowing no more than three requests in the *Hub*. Hence, in the *System* scenario, if we set $I_1 = 3$, and $I_2 = 1$, and include the restriction that $I_1 + I_2 \leq 3$, then, the possible scenario combinations are those depicted in Table 3.2.

Step 3.2: Concurrency-Level Models Generation

The next step is to generate a concurrency-level model for each child scenario $Scen_j$. Specifically, a concurrency-level model is a CTMC model, representing the number of completed instances of $Scen_j$, and the determination of the End state depends on the number of unique child scenarios. When there is one child scenario, completing I_j instances of $Scen_j$ represents the completion of the PAR scenario. i.e., the state I_j in the concurrency-level model is considered to be the End state. For example, the concurrency-level models corresponding to *Sensors_PAR* and *GWACK_PAR* in MIDAS are depicted in Figure 3.12(a) and (b).

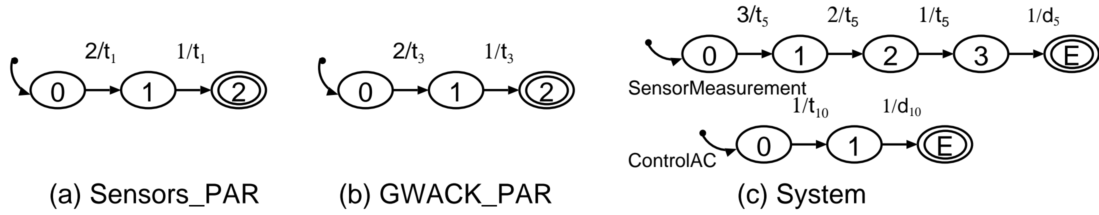


Figure 3.12: SBMs of the PAR scenarios

When there is more than one child scenarios, completing I_j instances of $Scen_j$ means that the execution of all instances has been completed. $Scen_j$ can only execute again when all other scenarios have been completed, and the parent scenario executes again. We add a state `E` to model this behavior, and define State `E` to be the End state of the concurrency-level model. We also add a transition from state I_j, which corresponds to completing all instances of $Scen_j$, to State `E`. For example, the concurrency-level models corresponding to the *System* scenario is depicted in Figure 3.12(c).

We determine the transition rates as follows: the transition rate from State c_j ($c_j < I_j$), i.e., when there are c_j completed instances of $Scen_j$, to State $c_j + 1$ is $(I_j - c_j)(1/t_j(1))$, where t_i is the scenario completion time, which is computed in Chapters 3.3.1 and 3.3.2. The transition rate corresponds to the rate an instance of $Scen_j$ completes, when there are c_j completed instances. When there are more than one unique child scenarios, the transition from State I_j to State `E`, which corresponds to the average time to wait for the completions of all instances from other scenarios, is $1/d_j$, where d is the average of the total delay other scenarios $Scen_k$, $k \neq j$ have caused. Here, we set $d_j = \sum_{k \neq j} I_k t_k$, where t_k is the completion time of $Scen_k$.[7]

[7]Note that d_j is an approximation, which assumes that the average time to complete an instance of $Scen_k$, given that $Scen_j$ has completed, is still t_k. An exact computation of d_j involves transient analysis, which is computationally more expensive [72].

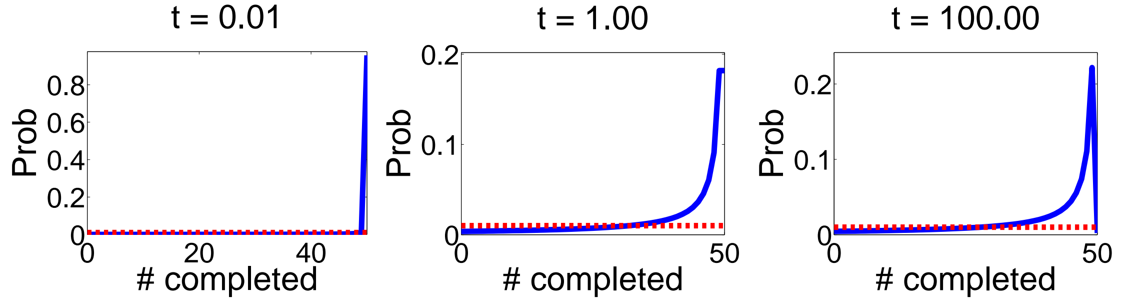


Figure 3.13: Probability distribution of the number of completed instances

Step 3.3: Performing Model Truncation

To further reduce the computational cost, we drop the combinations in a PAR scenario that are rarely visited using model truncation [69].

The steady probability distribution of c_j, the number of completed instances of $Scen_j$, depends on the values of $\vec{t_j}$ (scenario completion time), as well as the completion time of other scenarios $\vec{t_k}$, $Scen_k \neq Scen_j$. $P_j(c_j)$ can be obtained by solving a concurrency-level model using standard techniques. As an illustration, we depict the steady state probability distribution of c_j in Figure 3.13. We assume the completion rate of other scenarios are fixed, and $d_j = 1$ (recall Step 3.2). Also, we set $\mathbf{I} = 50$, and varied $t_j(1)$ at different values (0.01, 1, and 100). For instance, when $t_j(1) = 100$, 29 (out of 51) possible values of $P_j(c_j)$ is less than 1%.

In generating the scenario combinations, we can drop the values of c_j that occur rarely. That is, instead of considering c_j could be any value between 0 and $\mathbf{I}$, we consider a smaller range of values, determined as follows: in generating the scenario combinations, we consider x as a possible value of c_j if $P(c_j = x)$ is larger than a threshold ϵ. That is, a small threshold allows us to consider a wider range of value, and we can consider the case without using truncation as having $\epsilon = 0$. For example, if $\epsilon = 0.01$

(depicts as a dotted line in Figure 3.13), when $t = 100$, $d_j = 1$, and $\mathbf{I} = 50$, we only consider $29 \leq c_j \leq 50$ in generating the scenario combinations (instead of $0 \leq c_j \leq 50$).

There is a tradeoff between the number of states we drop and the penalty in accuracy in applying model truncation. We evaluate this tradeoff in Chapter 3.4.2.

Step 3.4: Computing Combination Probability

Once constructed, SHARP solves the concurrency-level model for the probability distribution of each combination. We define $P(C_i) = P(c_1, c_2, \ldots, c_{\mathbf{S}})$ to be the probability that there are c_j completed instances of $Scen_j$ for each $j = 1 \ldots \mathbf{S}$. Since we assume all instances of all child scenarios run independently,

$$P(C_i) \simeq \frac{\prod_j P_j(c_j)}{W} \tag{3.9}$$

where $P_j(c_j)$ is the probability that c_j instances of $Scen_j$ have completed, and $W = \sum_k P(C_k)$ is a normalization factor[8] that ensures that $P(C_k)$ sum to 1. In the *System* scenario, $P(c_1, c_2)$ is the probability that there are c_1 and c_2 completed instances of *SensorMeasurement* and *ControlAC*, respectively. Hence,

$$P(c_1, c_2) \simeq \frac{P_1(c_1) \times P_2(c_2)}{W}$$

We redistribute the transition state from the End state to State 1 as in Chapter 3.3.1, and solve the model for $P_j(c_j)$ for all c_j, using standard techniques [69]. This corresponds to solving for the probability of being in state c_j in $Scen_j$'s concurrency-level model. Table 3.3 gives the probability distribution of $P_j(c_j)$ in the two PAR scenarios of our MIDAS example; these are computed using the concurrency-level models of each

[8]The normalization factor is needed because, in general, not all combinations of scenarios may be allowed, as described earlier.

Table 3.3: Values of $P_j(c_j)$ in the *System* scenario

Parameter	Value	Parameter	Value	Parameter	Value
$P_1(0)$	0.0305	$P_1(2)$	0.0916	$P_2(0)$	0.8949
$P_1(1)$	0.0458	$P_1(3)$	0.8321	$P_2(1)$	0.1051

scenario (as described above). Furthermore, Table 3.2 gives the corresponding combination probabilities, computed using the data from Table 3.3 by applying Eq. (3.9). Lastly, since the computation of the distribution of different scenario combinations is done in an approximate manner as described above, in Chapter 3.4 we evaluate the accuracy of this approximation, as well as the reduction in computational cost.

At the level of the entire system (i.e., the highest level in the hierarchy), the synchronization requirement, that a PAR scenario is considered completed only when all child scenarios have been completed, may be too restrictive. Some systems are considered to be continuously running, where child scenarios can be considered as starting and completing independently. For example, after the *Sensors* have finished taking measurements, they do not necessarily have to wait until the *GUI* to update the data before taking another set of measurements. We can use our previous work in [18] to model this behavior.

Step 3.5: Computing Combination Reliability

Given that we now know how to compute the probabilities of having various combinations of child scenarios as well as the reliabilities of the child scenarios, what remains is the computation of the reliabilities of each combinations. We can then compute scenario reliability by combining the reliabilities of each combination. We will use the combination (1,0) in the *System* scenario — one completed instance of the *SensorMeasurement* scenario, and no completed instance of the *ControlAC* scenario — as an illustrative

example in this section; the reliabilities of other combinations are calculated analogously. To compute the reliability of a scenario combination, we need to first examine how scenario failure is defined.

In SHARP, system designers can specify the conditions under which the scenario is considered to have failed as follows. If there are x_j or more failed instances of *any* $Scen_j$, the system is considered to have failed, i.e.,

$$(F_1 \geq x_1) \vee (F_2 \geq x_2) \vee \ldots \vee (F_{\mathbf{S}} \geq x_{\mathbf{S}}) \tag{3.10}$$

where F_j is the number of failed instances of a child scenario $Scen_j$ (recall that $\mathbf{S}$ is the number of distinct child scenarios).[9] As an example, the system is considered reliable if it can control the temperature appropriately and display the current room temperature to the user. This requires that (a) *Sensors* on at least one *Gateway* correctly measure and send data to the *Hub*; and (b) the *GUI* displays the current temperature obtained from the *Hub*, and the *Hub* controls the *AC* appropriately. Therefore, we define the system to be unreliable when one or more instances of *SensorMeasurement*, or one or more instance of *ControlAC* have failed, respectively. Thus, $x_1 = 1$, and $x_2 = 1$.

To compute the probability that combination C_k satisfies the failure condition in Eq .(3.10), we consider the clauses one at a time, and compute the probability of satisfying a clause as follows:

$$P(F_j \geq x_j) = \sum_{f=x_j}^{I_j - c_j} P(F_j = f) \tag{3.11}$$

The next step is to compute the probability distribution of F_j, which is the number of failed instances of $Scen_j$, out of $I_j - c_j$ instances. An instance fails with probability

[9]The OR-clauses are used for ease of presentation. SHARP can easily specify more general failure conditions, by using disjunctive normal form and modifying Eq. (3.13) accordingly.

$1 - r_j$, where r_j is the reliability of $Scen_j$ as defined in Eq. (3.1); otherwise, with probability r_j, it has not failed. Note that F_j is a binomial random variable, and its probability distribution, according to [69], is as follows:

$$P(F_j = f) = \binom{I_j - c_j}{f}(1 - r_j)^f (r_j)^{(I_j - c_j - f)} \tag{3.12}$$

In the MIDAS example, based on Eq. (3.11) and Eq. (3.12), we compute $P(F_1 \geq 2)$ as follows:

$$\begin{aligned} P(F_1 = 1) &= \binom{2}{1}(1 - 0.9905)^1(0.9905)^1 = 0.0188 \\ P(F_1 = 2) &= \binom{2}{2}(1 - 0.9905)^2(0.9905)^0 = 9 \times 10^{-5} \\ P(F_1 \geq 1) &= \sum_{f=1}^{2} P(F_i = f) \\ &= P(F_1 = 1) + P(F_1 = 2) = 0.0188 \end{aligned}$$

Similarly, $P(F_2 \geq 1) = 0.0001$.

Since the system is considered to have failed when any clause in Eq. (3.10) is satisfied, the reliability of a combination, $R(C_k)$, can be defined as:

$$R(C_k) = 1 - \sum_{j=1}^{\mathbf{S}} P(F_j \geq x_j) \tag{3.13}$$

To complete our example, we combine the above results according to Eq. (3.13), i.e.,

$$
\begin{aligned}
R((1,0)) &= 1 - \sum_{i=1}^{\mathbf{S}} P(F_i \geq x_i) \\
&= 1 - (0.0188 + 0.0001) = 0.9811
\end{aligned}
$$

We repeat this calculation for each combination; Table 3.2 gives reliabilities of all scenario combinations for the MIDAS example under the above given failure conditions.

Step 3.6: Computing Scenario Reliability

We compute scenario reliability by combining the results of the previous steps. Scenario reliability of a PAR scenario is defined as the sum of the scenario combinations' reliabilities, weighted by the probability that the combination occurs, i.e.,

$$
r_i = \sum_k P(C_k) R(C_k) \tag{3.14}
$$

In our running example, the solution of Eq. (3.14) gives the reliability of the *System* scenario, and hence system reliability, as 0.9940, which, in this case, is within 0.5% of the ground truth of 0.9935, obtained by solving the "flat model" as detailed below.

Step 3.7: Computing Completion Time

To complete the PAR scenario analysis, SHARP computes the completion time for each combination C_j. Intuitively, the average completion time of a combination $t(C_j) = max(T_i(c_i))$, where $T_i(c_i)$ is a random variable representing the completion time of $Scen_i$ when there are c_i completed instances. Computing this is difficult, because we need to compute the completion time distribution of all child scenarios, followed by computing the distribution of $max(T_i(c_i))$.

To simplify the calculation of $t(C_j)$, we assume the completion of a scenario instance is memoryless, as in the flat model. As an example, consider the combination (0,0) in the concurrency-level models of the *System* scenario in Figure 3.12. i.e., no instances of both the *SensorMeasurement* and *ControlAC* scenarios have been completed. The memoryless assumption means that if an instance of the *SensorMeasurement* scenario has completed, the average time it takes to complete an instance of the *ControlAC* scenario is "reset" to t_{10}.

Given this assumption, we can see that the average time to complete a combination, $t(C_j)$, is simply the sum of the completion time for each scenario, $t_i(c_j)$. That is, the completion time for C_j is

$$t(C_j) = \sum_{i=1}^{\mathbf{S}} t_i(c_i) \tag{3.15}$$

To compute $t_i(c_i)$, we solve the concurrency-level for the average completion time, by setting State I_i as the End state of the concurrency-level model, and applying Eq. (3.3) to it. For example, in *Sensors_PAR*, we solve the model in Figure 3.12(a) using Eq. (3.3), and the resulting completion time is $T_i = [9.3937, 6.2625]$, and hence $t_i = 9.3937$.

3.4 Evaluation

We evaluate SHARP along two dimensions: (a) the complexity of generating and solving concurrent systems' reliability models as compared to those that can be derived from existing approaches (Chapter 3.4.1), and (b) the corresponding accuracy of SHARP (Chapter 3.4.2). More specifically, we compare SHARP against a *flat model*, which is used here as the "ground-truth". Our flat model is essentially the same as [59], where

a system reliability model is generated by applying the parallel composition.[10] We note that the difference between our application of parallel composition (in Chapter 3.3.1) and that in [59] is that we use parallel composition to generate a SBM of basic scenarios (which, as argued below, is expected to be relatively small) while [59] uses it to generate a model of the entire system at once.

We applied SHARP to a variety of systems, with different numbers of components, scenarios, as well as numbers of instances of scenarios. We show representative results obtained from the following systems:

1. The MIDAS example system we used throughout this chapter. This system has five basic scenarios, and may potentially have a large number of instances of a scenario (e.g., multiple sensors taking measurements). There are four *Sensors*, one *Gateway*, one *Hub*, one *GUI*, and one *AC* in the instantiation of MIDAS used in this evaluation.

2. A GPS system with route guidance, audio player, and bluetooth phone capabilities. This system has five major components: *RouteGuidance (RG)*, *EnergyMonitor (EM)*, *MediaPlayer (MP)*, *BluetoothPhone (BT)*, and *Database (DB)*. This system is modeled using 21 basic scenarios. Note that it is unlikely that there will be more than one instance of a scenario in this system because of the system's structure (e.g., it typically makes little sense to have two instances of a route guidance scenario to perform the same route guidance service).[11] To evaluate SHARP in a controlled manner, we injected the following defects into this GPS system: (a) a defect in the *EM* component which may lead to failure to notify other system

[10] The only differences between our flat model and the one in [59] is that [59] assumes that failures are irrecoverable. As we discussed earlier, SHARP can model irrecoverable failures without significant changes.

[11] One exception to this would be the situation when the system designers are concerned about service failures, and hence introduce redundancy. We do not consider such a variant of the GPS system.

components when the battery is low, and (b) a defect in the *RG* component which may lead to failure in updating a user's location accurately.

3. A simple client-server system with one server and possibly many clients, which is modeled using one basic scenario. In this system, a client sends a *Request* message to the server to request for service, and the service replies with a *Reply* message. If a request arrives when the server is servicing another request, the new request waits in the server's buffer in a FCFS fashion. We assume the server has enough buffer space so that the requests would not be dropped. We select such a simple system as it is difficult to study the accuracy of larger systems: the flat models of larger systems would become too large to solve, and hence we would lack a baseline for comparison. We use this system to illustrate the effect of contention when there are many clients, as its simplicity allows us to generate flat models with a larger number of components as compared to MIDAS (see Chapter 3.4.2 for details). We consider a defect in the server that may lead to failure to reply to the client when a requested file cannot be retrieved.

3.4.1 Complexity Analysis

We now explore the complexity of SHARP as compared to the flat model. We first describe the theoretical worst-case complexity of each approach, and then discuss the computational cost that is likely to arise in practice.

Worst Case Complexity

Let $\mathbf{U}$ be the number of unique components, $\mathbf{C}$ be the total number of components, $\mathbf{S}$ be the number of scenarios (basic and intermediary), $\mathbf{S_B}$ be the number of basic scenarios, $\mathbf{S_I}$ be the number of intermediary scenarios (i.e., $\mathbf{S} = \mathbf{S_B}+\mathbf{S_I}$). Also let $\mathbf{I}_i$ be the number

Table 3.4: Worst-case complexities

	Time Complexity	**Space Complexity**
SHARP	$O(\mathbf{S_I}\max(\mathbf{S^3}, \mathbf{S^{(I+1)}I^3})+$ $\mathbf{S_B}(\mathbf{M^{3C}}+\mathbf{M^C S_B(I+1)^{S_B}}))$	$O(\mathbf{S}+\max(\mathbf{M^{2C}}, \mathbf{S^2}, \mathbf{SI}))$
Flat Model	$O(\mathbf{M^{3C}})$	$O(\mathbf{M^{2C}})$

of instances of $Scen_i$, and $\mathbf{I} = max(\mathbf{I}_i)$ for all $Scen_i$. Also, let $\mathbf{M}_j$ be the number of states of $Comp_j$, and $\mathbf{M} = max(\mathbf{M}_j)$ for all $Comp_j$. The resulting complexities are summarized in Table 3.4. Let us first analyze the complexity of SHARP:

Basic scenarios: In the worst case, every state in every component participate in a basic scenario, and hence the SBM may have as many as $O(\mathbf{M^C})$ states. Once we have determined the states in the SBM, we need to determine the transitions between each pair of states. Therefore, the complexity of the generation of a SBM is $O(\mathbf{M}^{2\mathbf{C}})$. The complexity of solving a SBM [12] is $O(\mathbf{M}^{3\mathbf{C}})$. Thus, the time complexity of generating and solving the SBM for a basic scenario is $O((\mathbf{M^{2C}}+\mathbf{M^{3C}})) = O(\mathbf{M^{3C}})$. The space complexity of generating and solving a basic scenario's SBM is $O(\mathbf{M^{2C}})$ — once we have solved a SBM, we can reuse its space as we generate SBMs one at a time.

Contention Modeling: In the worst case, there is contention in every state in the SBM of a basic scenario. If, as a result, we add a queueing state corresponding to each state, we double the size of every SBM of each basic scenarios, which does not affect the worst case complexity of solving it ($O((2\mathbf{M^C})^3 = O(8\mathbf{M^{3C}}) = O(\mathbf{M}^{3\mathbf{C}})$). Thus, in the worst case, we have $O(\mathbf{M^C})$ QNs to solve. Determining the worst case complexity of solving a QN can be complicated, as that depends on the type of a QN we have. Given the special structure of our contention models (refer to Chapter 3.3.2), we can make sure that the corresponding QNs have product form [69], by adjusting the visit ratios

[12] The time complexity of solving a Markov chain with $\mathbf{N}$ states is $O(\mathbf{N^3})$, and the space complexity for storing the corresponding rate matrix is $O(\mathbf{N^2})$.

accordingly;[13] this would result in the worst case complexity of $O(\mathbf{S_B}(\mathbf{I}+\mathbf{1})^{\mathbf{S_B}})$ [69] for solving one QN. Thus, the worst case time complexity of solving all QNs would be $O(\mathbf{M^C}\mathbf{S_B}(\mathbf{I}+\mathbf{1})^{\mathbf{S_B}})$.

SEQ scenarios: Since there are at most $\mathbf{S}$ scenarios in the system, there are at most $\mathbf{S}$ states in the SBM of a SEQ scenario, because each scenario is represented by a state in the SBM of a SEQ scenario. Therefore, the complexities of generating and solving the SBM of a SEQ scenario are $O(\mathbf{S}^2)$ and $O(\mathbf{S}^3)$, respectively, and the space complexity is $O(\mathbf{S}^2)$, as discussed above.

PAR scenarios: In the worst case, all $\mathbf{S}$ scenarios run in parallel. The most expensive step in solving for $P(C_j)$ is to solve a concurrency-level model for each scenario (each with at most $O(\mathbf{I})$ states). Therefore, the complexity of solving for $P(C_j)$ is $O(\mathbf{SI^3})$. Solving for $R(C_j)$ involves combine the reliabilities of the child scenarios according to the failure conditions. The complexity of solving Eq. (3.11) is $O(\mathbf{I})$, and hence the complexity for computing $R(C_j)$ using Eq. (3.13) is $O(\mathbf{SI})$. Therefore, the complexity of solving for reliability of a PAR scenario is $O(\mathbf{S^I}(\mathbf{SI^3}+\mathbf{SI})) = O(\mathbf{S^I}\mathbf{SI^3}) = O(\mathbf{S^{I+1}}\mathbf{I^3})$. The space complexity of solving the PAR scenarios is $O(\mathbf{SI})$, as we need to store the results of the concurrency-level models.

Note that we have not considered the computational cost savings of model truncation in this complexity analysis, as model truncation does not reduce the worst-case complexity.

Overall Complexity: First, since there are $\mathbf{S_B}$ basic scenarios, the complexity of generating and solving all $\mathbf{S_B}$ SBMs of the basic scenarios is $O(\mathbf{S_B}(\mathbf{M^{3C}}+\mathbf{M^C}\mathbf{S_B}(\mathbf{I}+\mathbf{1})^{\mathbf{S_B}}))$. There are $\mathbf{S_I}$ intermediary scenarios, which each

[13] The visit ratios correspond to the the number of times the "callee queue" (e.g., the *Gateway* queue in Figure 3.8) is visited, per visit to the "caller queue" (e.g., *Sensor* queue in Figure 3.8). In our example, these are 1:1, but in general, the "callee queue" can be visited multiple times, per visit to the "caller queue".

of them could either be s SEQ or PAR scenario. As we do not know which of SEQ or PAR scenario is more expensive to solve in the worst case (it depends on the values of $\mathbf{S}$ and $\mathbf{I}$), we describe the complexity of solve an intermediary scenario to be $O(max(\mathbf{S^3}, \mathbf{S^{(I+1)}I^3}))$. Therefore, the overall time complexity is $O(\mathbf{S_B}(\mathbf{M^{3C}} + \mathbf{M^C S_B}(\mathbf{I}+\mathbf{1})^{\mathbf{S_B}})) + \mathbf{S_I} max(\mathbf{S^3}, \mathbf{S^{(I+1)}I^3}))$.

In analyzing the overall space complexity, we need to consider the space needed to store the results of the scenarios that have been processed, in addition to the space needed to store the SBM of the scenario that is being processed. Since we store the r_i and t_i of each $\mathbf{S}$ scenario in the worst case, the space needed to store the results of $\mathbf{S}$ scenarios is $O(2\mathbf{S})$. The "last" scenario could be a basic, SEQ, or a PAR scenario, so the space complexity is the maximum space needed among the three types of scenarios. Thus, the overall space complexity is $O(\mathbf{S} + max(\mathbf{M^{2C}}, \mathbf{S^2}, \mathbf{SI}))$.

In the flat model, we first apply parallel composition using all components, for which the complexity is $O(\mathbf{M^{2C}})$. The time complexity of solving the flat model is $O(\mathbf{M^{3C}})$. Therefore, the overall time complexity of the flat model approach is $O(\mathbf{M^{2C}} + \mathbf{M^{3C}}) = O(\mathbf{M^{3C}})$. Since the flat model has as many as $O(\ \mathbf{M^C})$ states, its space complexity is $O(\mathbf{M^{2C}})$.

Computational Cost in Practice

In our worst-case analysis above, it appears as if the flat model has the better time complexity. This is (partly) because in solving the SBM of the basic scenarios in SHARP, in the worst case, there are $\mathbf{M^C}$ states in each SBM (i.e., just as in the flat model).

However, in practice the worst case will be very unlikely. Specifically, the worst case analysis assumes that all states in all components participate in all scenarios. In practice, we expect that (a) the number of states participating in a scenario from a particular component, as well as (b) the number of components participating in that scenario, will

Table 3.5: Summary of computational costs in practice

	$\mathbf{U}$	$\mathbf{C}$	$\mathbf{M}$	$\mathbf{S_B}$	Flat Model	SHARP
MIDAS (6 sensors)	5	12	5	5	1.52×10^{12}	692
GPS	5	5	17	21	1.17×10^{11}	1331
Client-Server (8 clients)	2	9	2	1	1.34×10^{8}	737

be substantially smaller than $\mathbf{M}$ and $\mathbf{C}$, respectively. In contrast, even in practice, the flat model approach requires generation of the entire system model that involves using all states in all components. Thus, we expect the worst case analysis of the flat model approach to be reflective of the practice.

Another reason for the increased worst case complexity of SHARP is the assumption that all scenario types participate in all resource contention points, which leads to more costly solutions of QNs, used for contention modeling. Again, in practice, we expect that the number of scenario types contending for the same resource would be substantially smaller than $\mathbf{S}$. Moreover, many approximation techniques exist in the QN literature, which based on our experience should work well, given the simple structure of our QNs. For instance, Schweitzer's approximation [64] would result in an $O(\mathbf{S})$ worst case solution.

Table 3.5 summarizes the computational costs in practice to solve for system reliability using the flat model and SHARP of the three systems we evaluated. The computational cost savings using SHARP are significant in all three systems. This illustrates that SHARP is able to avoid scalability problems by generating and solving many smaller models, instead of generating and solving one huge model as in the flat model. Comparing the computational costs of the three systems yields some interesting observations. We noticed that it is more expensive to solve the GPS system than the MIDAS systems, since the GPS system is modeled as 21 basic scenarios, and we generate and solve a SBM for each scenario. Although systems with more basic scenarios (which are typically more complex) are more expensive to solve in SHARP, the computational costs

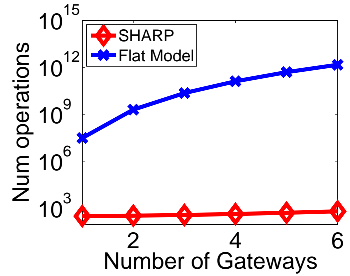


Figure 3.14: Computational Cost in Practice

in practice are still significantly lower, as compared to the flat model. While the client-server system is a simpler system than MIDAS, it costs more to solve. This is because $\mathbf{I}$ is larger in the client-server system ($\mathbf{I} = 8$) than MIDAS ($\mathbf{I} = 3$), which results in a larger model in a PAR scenario. The computational cost of the client-server system can be reduced using model truncation, as described in Chapter 3.3.3 (Step 3.3).

Figure 3.14 illustrates how computational costs increase as the system becomes more complex. Here, we vary the number of *Gateways* in MIDAS (x-axis), and assume that each *Gateway* connects to two *Sensors*. We plot the number of addition/multiplication operations needed to solve the two resulting models on the y-axis. Otherwise, the system is the same as the example used throughout this chapter. Note that the y-axis of Figure 3.14 is plotted on a logarithmic-scale. As can be seen from the figure, the computational cost of SHARP is much lower and grows significantly slower than that of the flat model. For example, it takes more than 10^{12} operations to compute the reliability solution of the MIDAS system with 6 *Gateways* using the flat model, while it only takes 692 operations to compute the solution using SHARP.

Since the SBMs are likely to be smaller than the flat model, we argue that SHARP requires significantly less space in practice than the flat model. The savings are also due to the fact that we can generate and solve the SBM one at a time, and thus reuse the space.

As we discussed in Chapter 3.3.3, it is also possible to reduce the computational cost of SHARP via model truncation. We study the tradeoff between further reducing the computational cost and loss of accuracy in Chapter 3.4.2.

Lastly, given that SHARP takes the approach of solving many smaller models rather than one large model, if parallel processing is available, we could solve our models in parallel.

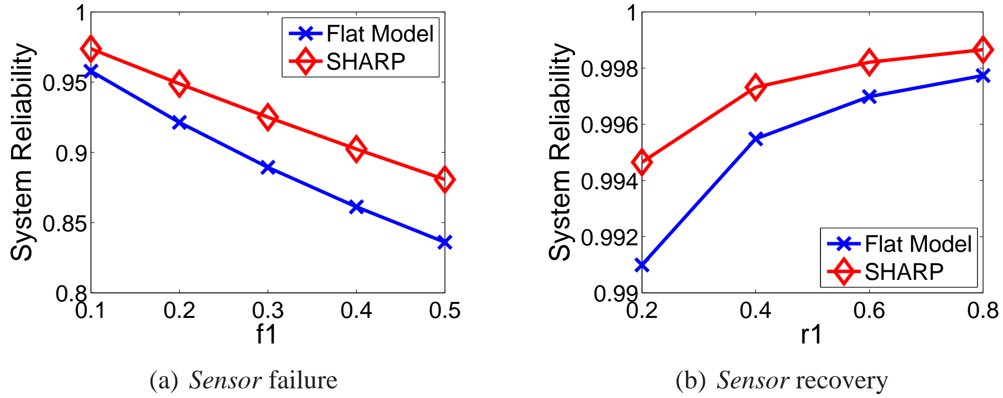
3.4.2 Accuracy

Our goal is to provide evidence that SHARP is sufficiently accurate to be used in making design decisions. The goal of design-time approaches is to analyze the effect of different design decisions on reliability rather than obtain absolute reliability measurements.[14] Therefore, we compare the *sensitivities* of SHARP and the corresponding flat model: if the differences in the change in reliability estimates are reasonably small when the same parameter is varied in both SHARP and the flat model, then SHARP can be considered accurate.

Sensitivity Analysis

First, we compare the sensitivities of SHARP and the flat model when model parameters change. We vary a parameter within a range (to be specified below), and observe how system reliability changes. Here, we present results corresponding to varying failure-related parameters in the MIDAS and GPS systems. We performed similar experiments by varying other parameters and using several other systems' models. The results were qualitatively similar and are omitted here for brevity.

[14]For example, at implementation time, it may be appropriate to evaluate a system's reliability using the five 9's standard. However, this is not typically meaningful at design time.



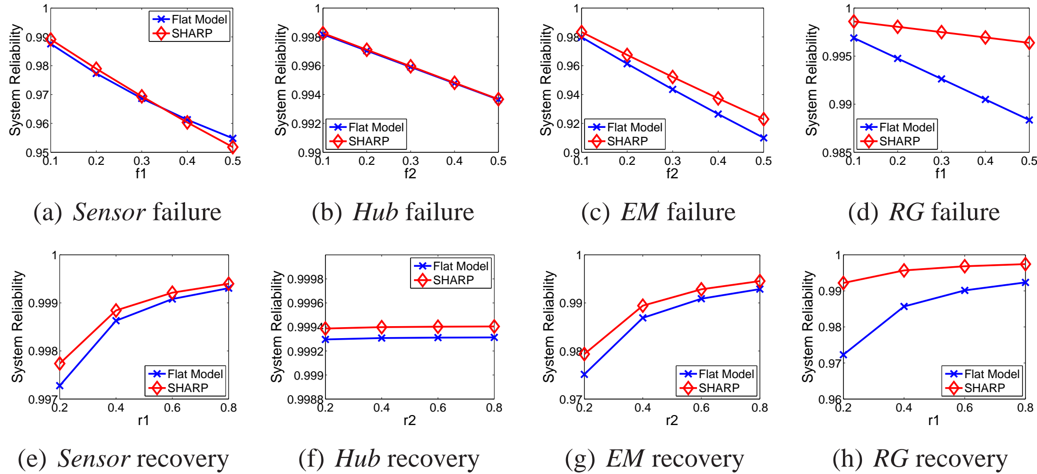
(a) *Sensor* failure (b) *Sensor* recovery

Figure 3.15: Sensitivity Analysis of the *Sensor_PAR* scenario

The inaccuracies in our estimates come from the solution of the PAR scenarios, because of the approximations we made (recall Chapter 3.3.3). We generate the SBM of the basic scenarios using the same technique as in existing techniques, therefore the results are the same. The solution of the SEQ scenarios is exact: the steady state probability using our stochastic complementation-based approach is the same as one would solve it directly (by "flattening out" the model and connect the child scenarios appropriately) [49].

We compare the sensitivity at the level of a scenario, and our results of the *Sensor_PAR* scenarios are depicted in Figure 3.15. We varied the failure rate of the two *Sensors* between 0.1 to 0.5 in Figure 3.15(a) and the recovery rate between 0.2 and 0.8 in Figure 3.15(b). We vary the parameters one at a time, maintaining other parameters fixed at their default values (Default values of the MIDAS system are given in Figure 3.2). We varied other parameters in the *Sensor_PAR* scenario, and the results are qualitatively similar.

As we can see, the reliability estimates of both SHARP and flat model vary at a similar rate when the parameters change. For example, in Figure 3.15(a), scenario reliability drops from 0.96 to 0.87 in SHARP, and from 0.95 to 0.84 in the flat model, respectively.



(a) *Sensor* failure (b) *Hub* failure (c) *EM* failure (d) *RG* failure

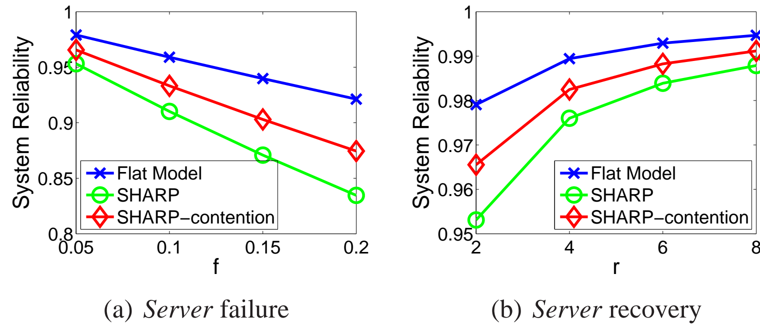
(e) *Sensor* recovery (f) *Hub* recovery (g) *EM* recovery (h) *RG* recovery

Figure 3.16: Sensitivity analysis at the system level

Next, we study how the inaccuracies propagate to the system level. In Figures 3.16(a) - (d), we vary the failure rates of the *Sensor* and *Hub* components in MIDAS, and the *EM* and *RG* components in GPS between 0.1 and 0.5. In Figures 3.16(e) - (h), we vary the recovery rates of the *Sensor*, *Hub* components in MIDAS, and the *EM* and *RG* components in GPS between 0.2 and 0.8, As in the experiments at the level of a scenario, other parameters fixed at their default values.

In these experiments, we observe that results obtained from SHARP closely follow the flat model. This suggests that SHARP is accurate in predicting system reliability, while in practice it should result in much better scalability than the flat model approach.

We also illustrate that SHARP is useful in determining which components are more critical to a system's reliability. We have verified this property of SHARP in a number of examples. For instance, in Figure 3.16, when we vary the failure rates of *Sensor* (Figure 3.16(a)) and *Hub* (Figure 3.16(b)) between 0.1 and 0.5, system reliabilities obtained from SHARP change by 4% and 0.4%, respectively. Since the system's reliability is affected more by the changes in *Sensor*'s failure rate than *Hub*'s, under these conditions *Sensor* is the more critical component. Note that the differences in the change in



(a) *Server* failure (b) *Server* recovery

Figure 3.17: Sensitivity analysis of the Client-Server system

reliability estimates between SHARP and the flat model are very small (within a few percent). We have not observed significant deviations from the flat model in any of our studies. We can thus conclude that SHARP is useful in this analysis.

Effect of Contention Modeling

This section aims at illustrating the importance of modeling contention in SHARP. We use a very simple client-server system with a single scenario for reasons given above. By increasing the number of clients, we can model a highly-contended system. For example, our results with one server and 8 clients are depicted in Figure 3.17, where we present the results of using SHARP without contention modeling, SHARP with contention modeling by considering the server as a FCFS callee, as well as results from the flat model (which includes contention) as a baseline for comparison. The differences between the results obtained from SHARP without contention modeling and the flat model can be as large as 12% (when the failure rate is 0.2), while the results with contention modeling are much more accurate (the error is generally about 2%, and no larger than 5%, when the failure rate is 0.2). This occurs because, without contention modeling, SHARP includes the time spent waiting to be served as processing time, thus

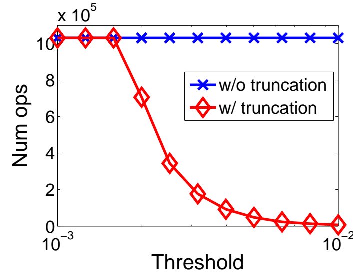


Figure 3.18: Computational cost of SHARP with and without truncation

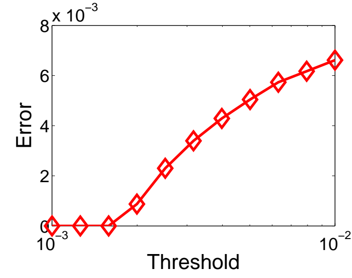


Figure 3.19: Errors caused by model truncation

overestimating the processing time. In turn, this lowers the reliability because processing a request may trigger a defect in the server that waiting for service does not. Results obtained using other systems are qualitatively similar, and are omitted for brevity.

Effect of Model Truncation

In evaluating the effect of truncation in Chapter 3.3.3, first we study the computational cost savings. In Figure 3.18, we plot the number of operations needed to solve for the reliability of MIDAS with one *GUI*, *AC*, and *Hub*, and vary the number of *Gateways*. As in Chapter 3.4.1, we assume that each *Gateway* connects to two *Sensors*, and increase the number of *Sensors* accordingly. The interactions of each *Gateway* with other components are modeled as an instance of the *SensorMeasurement* scenario. There are at most 100 instances of the *SensorMeasurement* scenario, and at most one instance of the *GUIRequest* scenario. In Figure 3.18, we varied the threshold (x-axis, plotted in logarithmic-scale), and plotted the number of operations needed to solve SHARP with truncation. We fixed the scenario reliability of *SensorMeasurement* at 0.99, and the completion time at 1. The system is considered to have failed when any instance of *SensorMeasurement* has failed. The cost without truncation is our baseline, and can be

considered as having a threshold of 0. As we can see from Figure 3.18, the computational cost savings can be significant: when the threshold is 10^{-2}, the number of operations needed to solve SHARP with and without truncation are approximately 1×10^6 and 6200, respectively. These results indicate that model truncation reduces the computationally cost in generating the scenario combinations.

Next, we study the error in reliability estimates when truncation is used (i.e., we consider only a small range of possible values of the number of active instances). The results are depicted in Figure 3.19. We varied the threshold in the x-axis, and plotted the error in reliability estimates as compared to the results without truncation (y-axis). When the threshold is small (i.e., we consider a wider ranges of values), the error is smaller, with the largest error being 0.8% (when the threshold is 10^{-2}).

3.5 Conclusions

We presented SHARP, a scalable framework for predicting reliability of concurrent systems. Our main idea in modeling concurrency is to allow multiple instances of system scenarios to run simultaneously. We overcame inherent scalability problems by leveraging scenario models and using an (approximate) hierarchical technique which allowed generation and solution of smaller parts of the overall model at a time. Our experimental evaluation showed that SHARP is more scalable than existing approaches in practice, and its scalability is achieved without significant degradation in the accuracy of system reliability predictions.

Chapter 4

Performance Estimation of Third-Party Components

As discussed earlier, it is expensive to apply testing-based approaches to assess the quality of software systems. To address this problem in the context of performance estimation, we propose a queueing model-based framework that estimates software performance at high workloads, by applying regression analysis using performance testing data collected at low workloads. We focus on applying this framework to estimating the performance for Web services (WSs), because, as discussed in Chapter 1, software designers have to rely on testing-based approaches to evaluate the quality of software in the "Binaries Accessible" category, and WS is one such example.

An overview of our framework is depicted in Figure 4.1. In Step 1, we collect performance data of the WS being tested using performance testing. In Step 2, we apply regression analysis to estimate response time at points that are not sampled during performance testing, using data collected in Step 1. Our main contribution is in Step 2, where we propose incorporation of queueing models in this process, in order

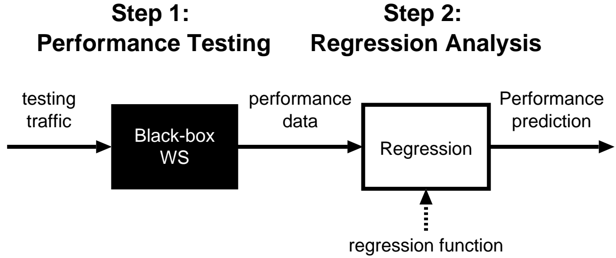


Figure 4.1: An overview of our WS performance prediction framework

to overcome the poor extrapolation results typically obtained using standard regression analysis-based techniques (as detailed below).

4.1 A Framework for WS Performance Prediction

We describe our framework in details in this section. Specifically, in Step 1, we send requests to the WS being tested, and collect the corresponding average response time. This process is repeated at different workload intensities. Performance testing (Step 1, Chapter 4.1.1) is typically done to ensure that the system of interest conforms to some performance expectations. The challenge in this step is that performance testing is quite expensive, as it involves making a large number of requests to the WS being tested. This is especially the case for testing under heavy loads, where the testing process can greatly affect the normal operation of the WS being tested. The implication then is that we have limited data, particularly for the system under heavy loads, to predict the WS performance. Thus, it is highly desirable to be able to *extrapolate* – using data collected under lighter loads to construct predictive models which are capable of predicting well under heavier loads.

In other words, Step 2 (Chapter 4.1.2) involves predicting response time *beyond* the sampled arrival rates in Step 1. Extrapolation is a challenging problem, and we have confirmed that standard regression approaches perform poorly at this task (refer to Chapter 4.1.2). Therefore, we propose to use queueing models for response time prediction, which, however, may give less accurate interpolation results. This motivates us to derive a hybrid approach that combines the more accurate interpolation results when using standard regression approaches, with the more accurate extrapolation results when using queueing models.

4.1.1 Step 1: Performance Testing

Performance testing has been used in evaluating software performance to ensure the system performs as expected [51]. The goal of performance testing is to understand the system's properties, such as system throughput and response time, given a controlled workload.

Performance testing may assume an open model, in which clients arrive to the system at a pre-specified arrival rate $\hat{\lambda}_E$, and leave the system once the request has been served. It may also assume a closed model, in which the number of clients is fixed. In either case, we are interested in observing the response time when we vary the arrival rate in an open model, or when we vary the number of clients in a closed model.

In the remainder of this chapter, we assume the use of an open model, and generate arrivals accordingly to a Poisson process. (We note that, as a client of a third-party WS, we can control the arrival process.) Specifically, we generate D^j requests at rate $\hat{\lambda}_E^j$, and measure the response time to each request k, $\hat{T}^{j,k}$. We can then compute the average response time as

$$\hat{T^j} = \frac{1}{D^j} \sum_k \hat{T^{j,k}} \tag{4.1}$$

We repeat the test at different values of $\hat{\lambda}_E^j$, and compute the corresponding $\hat{T}^j$.

A shortcoming of performance testing is the assumption that the system does not change over the duration of the test. This includes the WS being tested, any other third-party WSs involved, as well as network conditions. In real-world applications, this may not be the case. For example, making a large number of requests to a WS may be perceived as an attack. Thus, administrators may block the testing traffic, and, as a result, we would not be able to gather performance data. This again motivates the need to limit performance testing, particularly at higher workloads, and devise approaches for accurate extrapolation.

4.1.2 Step 2: Regression Analysis

The goal of regression analysis is to model and estimate the input-output relationship between random variables based on observed data, and then use the model for prediction. In our context, we apply regression analysis to model the relationship between the arrival rate and the WS response time, and predict WS response times at arrival rates that are not sampled during performance testing. The stage of modeling is often referred as "training". We often need to assess the effectiveness of a trained model before we deploy it to real-world environments to make prediction. The stage of assessment is often referred as "testing" (or "evaluation" to avoid being confused with performance testing). The assessment is accomplished by comparing the model's prediction on data with known arrival rates and responses times. However, such data should have no overlap with the data used in the training stage so that the model is not over-optimistic.

As noted earlier, we differentiate two different types of predictions: interpolation when the arrival dates are *within* the range of those being collected during performance testing, and extrapolation when the arrival dates are *outside* the range.

Statistical models for regression analysis can be broadly classified into *parametric* and *nonparametric*:

Parametric regression: In parametric regression, we specify a regression function with unknown parameters to capture the relationship between the arrival rate and the response time. One can leverage prior knowledge about the relationships among variables to determine a suitable regression function. An example regression function is an Nth-degree polynomial, i.e.,

$$T(\lambda_E, \vec{\alpha}) = \sum_{i=0}^{N} \alpha_i (\lambda_E)^i \tag{4.2}$$

where λ_E is the average customer arrival rate to the WS. $\vec{\alpha}$ is the unknown parameter vector, representing the coefficients of the polynomial. We estimate it using performance testing data.

More specifically, given a regression function and data from performance testing (pairs of values of $\hat{\lambda}_E^j$ and $\hat{T}^j$), we would like to find $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_K)$ such that the mean squared error between the measured response time and the model's prediction is minimized. This problem can be formulated as the following optimization problem:

$$\underset{\vec{\alpha}}{\text{minimize}} \quad \sum_j (\hat{T}^j - T(\hat{\lambda}_E^j, \vec{\alpha}))^2 \tag{4.3}$$

where $T(\hat{\lambda}_E^j, \vec{\alpha})$ is the predicted response time when the external arrival rate is $\hat{\lambda}_E^j$. This problem can be solved using standard optimization techniques [26].

In addition to fitting data with a polynomial, we have also considered fitting the data with an exponential function. i.e.,

$$T(\lambda_E, \vec{\alpha}) = \alpha_1 e^{\alpha_2 \lambda_E} \tag{4.4}$$

where $\vec{\alpha}$ is estimated using regression.

Once we have estimated the unknown parameters, we predict response time, by plugging in the arrival rate of interest, λ_E, and parameters estimated from regression, $\vec{\alpha}^*$.

In this work, we consider another two types of regression functions – splines and neural networks. Splines are piecewise-smooth polynomials. We used cubic splines in our experiments (a standard choice in many applications [25]).

Neural networks (NN) are another common approach for regression. Architecturally, a neural network is a set of connected linear and nonlinear "neurons". They can model highly nonlinear functions with sufficiently complicated network architecture. In our

experiments, we have used 3-layer neural networks. The first layer is the input layer, representing the arrival rate. The output layer corresponds to the response time. The hidden layer is a layer of nonlinear processing units which transform the input with $\tanh$ functions. The transformed inputs are then linearly combined to form the output [13].

Nonparametric regression: Nonparametric approaches make predictions directly utilizing the observed data, without specifying explicitly a regression function. An example of nonparametric approaches is Gaussian process (GP) [57]. In our work, the GP encodes similarity among data (i.e., pairs of arrival rates and response times) with kernel functions and makes predictions by combining (nonlinear) response times from observed data. Intuitively, a closer training data at $\hat{\lambda}_E$ to λ_E contributes more to the final prediction on λ_E. Our experiments use the so-called "neural network $\tanh$ kernel" as it performs the best when compared to a few other alternatives.

A Shortcoming of Standard Regression Analysis

To illustrate a shortcoming of applying standard regression analysis for WS performance estimation, we show how well these approaches extrapolate. A more comprehensive validation is presented in Chapter 4.2.

In this experiment, we use extrapolation error as our metric. We collected performance testing data by varying the arrival rates, until the system has been saturated (i.e., when the system has started returning errors because of resource saturation). We then divide the data into two sets: the training set and the validation set. Data in the training set, consisting of the data points in the bottom 60% of the arrival rates sampled, was supplied to the regression algorithm. Then, we compute the extrapolation error by comparing the predicted response time and data in the validation set, which corresponds to the data points in the upper 40% of the arrival rates sampled.

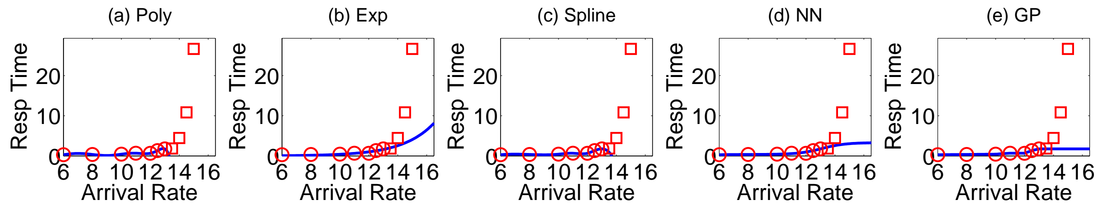


Figure 4.2: Extrapolation using standard regression methods

Here, we show the results using the Java Adventure Builder (AB) application [3]. This simple travel agent WS is provided by Sun to demonstrate the development and deployment of a WS. It is an atomic WS (i.e., one that does not make requests to other WSs), that makes requests to a local database server. Our system has 54 customers and 1,022 bookings.

The extrapolation results are depicted in Figure 4.2. Data in the training set and the validation set are depicted as circles and squares, respectively. We depict results based on an 8th-degree polynomial in Figure 4.2(a) as, in this experiment, the results of using an 8th-degree polynomial were more accurate than those using polynomials of other degrees. We observe, from Figure 4.2, that standard regression techniques are unable to predict response time when the arrival rates are outside of the data used as input to regression analysis. Specifically, all five approaches we studied predict the response time to remain flat when the arrival rate increases beyond the sampled arrival rates, instead of increasing rapidly as the system nears saturation. Indeed, the fact that standard regression approaches may give poor extrapolation results is a well-known problem in the regression literature.

A Queueing Model-Based Framework

To address the shortcoming that standard regression approaches tend to perform poorly at extrapolation, we propose a queueing network-based framework to estimate the response time of black-box WSs. More specifically, we use queueing models to derive

a function that describes the relationship between arrival rates and response time; this function is then used as the regression function in parametric regression for response time prediction. The challenge is, however, that we do not know the structure of the WS being tested. For example, we do not know if it is deployed on a server, using a three-tier architecture as in [74], or if it makes use of other WSs. In the absence of structural information, we approach this problem by using a suite of queueing models, and as shown in Chapter 4.2, this provides us with insight about the performance of the WS. For example, we can determine the stability conditions of the WS using the most pessimistic model.

In presenting our queueing model-based framework, we first discuss single-queue models, followed by queueing network models. We also give several instantiations of the queueing models we have considered in our evaluation in Chapter 4.2.

Single-Queue Models: A single-queue model is characterized by: (1) the *arrival process*, which describes the workload characteristics; (2) the *service time distribution*, which describes the characteristics of the servers; and (3) the *number of servers*, which describes the degree of concurrency. As a client to a WS, we can control the arrival process by adjusting the performance testing parameters. Parameters related to the service time distribution are estimated using regression, while the number of servers is determined by the system modelers. Given this information, we can derive the average response time as a function of arrival rate and other model parameters, and estimate model parameters by applying standard regression analysis using data collected from performance testing.

Since information about the WS being tested is limited, in general, it is challenging to determine the number of servers and the service time distribution. However, in our validation in Chapter 4.2, we show that even with simple queueing models (as detailed below) one can gain valuable insight into the WS being tested. For instance, we can

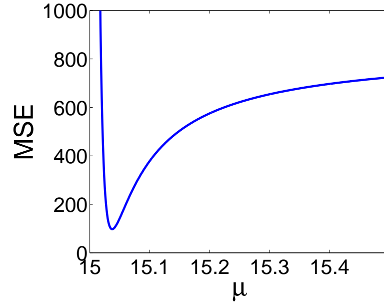


Figure 4.3: An Example Objective Function

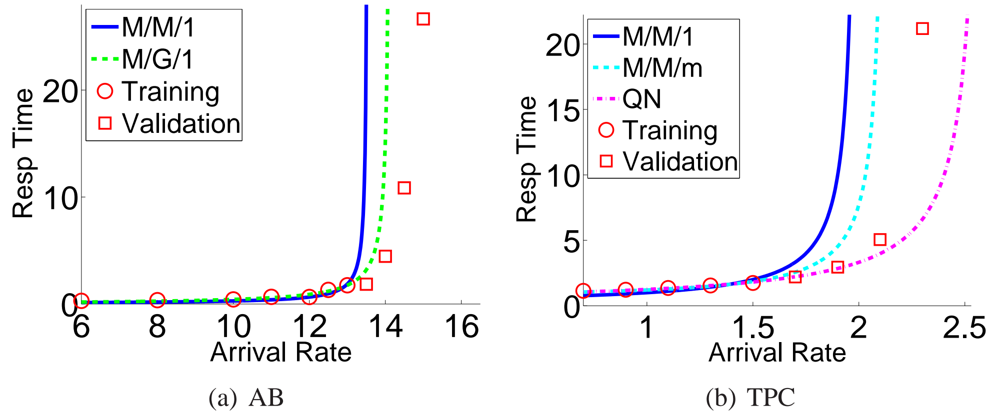
determine the stability conditions of the WS, which are useful in, for instance, determining how much workload one should send to that WS.

M/M/1 Model: As an example, let us consider the M/M/1 model (i.e., with a Poisson arrival process and exponential service time distribution). The corresponding average response time is then [69]

$$T(\lambda_E, \mu) = 1/(\mu - \lambda_E) \tag{4.5}$$

where λ_E and μ are the average customer arrival and service rates, respectively. We apply regression analysis to estimate μ using performance testing data. In applying regression analysis, we need to specify *constraints* to ensure that the resulting system is stable, i.e., in the case of the M/M/1 model, that $\mu > \lambda_E$.

Another important consideration is the choice of a *starting point* to the regression problem. It has been proven that regression algorithms can find the global optima when the objective function is convex, no matter which starting point we choose [26]. However, as opposed to using a polynomial as a regression function, using queueing models may result in a non-convex objective function. An example of our objective function is depicted in Figure 4.3, in which we apply the M/M/1 model to the AB WS. In this example, the highest arrival rates we used in performance testing, $\hat{\lambda}_E^{max} = \max_j \hat{\lambda}_E^j$, is 15, and we varied μ (x-axis) between 15.01 and 15.5. The MSE is plotted on the y-axis,



(a) AB (b) TPC

Figure 4.4: Extrapolation using queueing models

and it can be shown that this objective function is not convex. It appears that the objective function is convex when $\mu < \mu^*$, where μ^* is the optimal point that gives the lowest MSE (in this example, when $\mu \simeq 15.03$). Therefore, we set μ slightly larger than the highest arrival rates used in the performance, $\hat{\lambda}_E^{max}$ (we set μ to $1.001\hat{\lambda}_E^{max}$ in our evaluation). We have experimented with other starting points, and our results have indicated that our choice has yielded good results. We are unsure if our choice of a starting point is optimal; such analysis is out of the scope of this chapter.

We apply regression analysis to predict the response time of the AB WS using the M/M/1 model, with results depicted in Figure 4.4(a). Even though the M/M/1 model can predict the rapid increase in response time (beyond a certain load), it does so pessimistically in this case, i.e., this increase occurs much sooner than in the actual system. One reason for this is that the exponential service time distribution assumption is unlikely to hold in a real system. Thus, the M/M/1 model illustrates the basic idea and motivates the use of more complex models, as we do next.

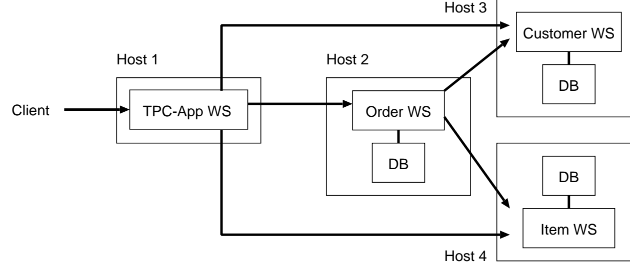


Figure 4.5: TPC WS Architecture

M/G/1 Model: The M/G/1 model allows a general service time distribution that is characterized by its mean and variance. The corresponding average response time is [69]:

$$T(\lambda_E, \mu, \sigma) = \frac{1}{\mu} + \frac{\lambda_E \sigma^2}{2(1 - \frac{\lambda_E}{\mu})} \tag{4.6}$$

where σ^2 is the variance of the service time distribution. We apply regression analysis to estimate both μ and σ using performance testing data.

The results of predicting response time of the AB WS using M/G/1 model is depicted in Figure 4.4(a). The M/G/1 model is more accurate than the M/M/1 model, due to the more general model of the service time distribution.

M/M/m Model: The M/M/m model relaxes the single-server assumption of the M/M/1 model, i.e., we have a single queue with m servers. The corresponding average response time is [69]:

$$T(\lambda_E, \mu, m) = P_Q \frac{\rho}{\lambda_E(1 - \rho)} + \frac{1}{\mu} \tag{4.7}$$

where $\rho = \lambda_E/(m\mu)$ and P_Q, the probability of queueing, is given by [69]:

$$P_Q = \frac{(m\rho)^m}{m!} \frac{p_0}{1 - \rho} \tag{4.8}$$

$$p_0 = (\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{(1 - \rho)})^{-1} \tag{4.9}$$

To illustrate the use of multi-server queueing models, we present results using the TPC-App benchmark [6] we deployed, which we refer to as TPC WS in the remainder of the chapter. This benchmark emulates a bookstore WS environment, in which customers can create an account, search for books, place an order, and retrieve the status of an order. Our deployment of TPC is depicted in Figure 4.5. The WS makes use of several internal WSs: an Order WS, an Item WS, and a Customer WS. Each of the three internal WSs runs on a separate physical machine, and queries a local database. Our system has 100 customers, 500 books, and 30,000 order records.

The extrapolation results[1] using the TPC WS are depicted in Figure 4.4(b). While the results based on the M/M/m model are more accurate than those based on the M/M/1 model, they are still pessimistic. One reason is that the TPC WS was deployed on four machines, and each machine has its own queue. Therefore, a single-queue, multi-server model, such as the M/M/m model, may not be as accurate as a model with multiple queues, which motivates consideration of queueing network models.

Queueing Network Models: To simplify our discussion, we assume an open[2] QN of M/M/1 queues. We also assume there is only one class of customers: the arrival and service processes for all customers are the same. In such a QN, a queue may, e.g., represent an internal server (such as a Web server or a database server), or another WS. With these assumptions, our QN is a product-form network [11].[3]

The first piece of information needed in addition to single-queue models is the *number of queues*, which is estimated by the system modelers. This corresponds to the number of physical servers (e.g., database and application servers) that serve a client's request. One approach is to try different number of queues, and determine which gives

---

[1]Here $m = 2$; different values of m gave less accurate results.

[2]A closed model can be used without significant changes to our approach.

[3]In general, more complex QNs can be used and still remain product-form [11]; We would then update Eqs. (4.12) – (4.15) to reflect that.

the most accurate results. We use a two-queue QN to model the TPC WS in Chapter 4.2, as it generates the most accurate results among QNs with different number of queues. For each queue, in addition to the parameters specified in single-queue models, we need to determine its *visit ratio*, using regression (see below).

We now define a QN model more formally. Let K be the number of queues, $p_{i,j}$ be the probability of going to queue j upon leaving queue i; $p_{E,i}$ be the probability that an external arrival goes to queue i, and $p_{i,E}$ be the probability that a customer leaves the system upon leaving queue i, where $\sum_j p_{i,j} + p_{i,E} = 1$. Note that in a WS, a customer always arrives at the WS being tested (e.g., a customer cannot send requests directly to an internal database server). If we assume Queue 1 is the WS being tested, then $p_{E,1} = 1$, and $p_{E,i} = 0$ for all $i \neq 1$.

The visit ratio of queue i is given by [69]

$$v_i = p_{E,i} + \sum_j v_j p_{j,i} \tag{4.10}$$

where the total arrival rate at queue i is

$$\lambda_i = \lambda_E v_i \tag{4.11}$$

Given λ_i for each M/M/1 queue i, the average number of customers in queue i, N_i, is [69]:

$$N_i = \frac{\lambda_i}{\mu_i - \lambda_i} = \frac{\lambda_E v_i}{\mu_i - \lambda_E v_i} \tag{4.12}$$

Since the QN is product-form, the joint probability of having n_i customers in queue i, $1 \leq i \leq K$, is

$$P(n_1, n_2, \ldots, n_K) = \prod_i P(n_i) \tag{4.13}$$

where $P(n_i)$ is the probability that there are n_i customers in queue i. Here, the average number of customers in the system, N, is

$$N = \sum_i N_i = \sum_i \frac{\lambda_E v_i}{\mu_i - \lambda_E v_i} \tag{4.14}$$

Thus, using Little's result [69], the average response time is

$$T = \frac{N}{\lambda_E} = \frac{1}{\lambda_E} \sum_i \frac{\lambda_E v_i}{\mu_i - \lambda_E v_i} = \sum_i \frac{v_i}{\mu_i - \lambda_E v_i} \tag{4.15}$$

We can simplify our process as follows: instead of estimating the entire routing matrix (i.e., the $p_{i,j}$'s) and compute the visit ratios, we choose to estimate the visit ratio v_i's directly. Furthermore, if we multiply Eq. (4.15) by $(1/v_i)/(1/v_i)$, we obtain:

$$T = \sum_i \frac{v_i}{\mu_i - \lambda_E v_i} \times \frac{1/v_i}{1/v_i} \tag{4.16}$$

$$= \sum_i \frac{1}{\mu_i/v_i - \lambda_E} = \sum_i \frac{1}{\alpha_i - \lambda_E} \tag{4.17}$$

where $\alpha_i = \mu_i/v_i$. Rewriting Eq. (4.15) as Eq. (4.17) allows us to simplify the response time estimation process by using regression analysis to estimate μ_i/v_i directly, instead of their individual values.

We apply this QN model to the data collected from the TPC WS, with the results depicted in Figure 4.4(b). We observe that the QN model is more accurate than the M/M/1 and M/M/m models, because of its more accurate description of the TPC WS's structure. This QN model, however, is too optimistic when the arrival rate is high. This suggests that we should use a suite of queueing models to understand the behavior of a WS, rather than a single model.

Table 4.1: Comparisons of TPC WS interpolation results

λ	measured	QN	error	NN	error
0.7	1.13700	1.03342	0.10358	1.14992	0.01292
1.1	1.35670	1.31543	0.04127	1.33618	0.02052
1.5	1.74110	1.82561	0.08451	1.78227	0.04117
1.9	2.94880	3.09797	0.14917	2.90909	0.03971

A Shortcoming of Queueing Models

While the extrapolation results using queueing models are better than those of standard approaches, their interpolation results are not as good. This can be explained as follows. System response time increases rapidly when the system is close to being saturated, and hence the slope of the response time function is very steep when λ_E is high. This property causes the regression algorithm to try fitting data at high workload intensity, because a slight error in the estimated parameters results in very large errors in these data points. Given that the queueing models usually have few parameters to fit (e.g., the M/M/1 model only has one parameter), the regression algorithm cannot adjust the parameters to fit data at low workload intensity, and hence the response time estimates at low workload intensity are not as good using queueing models. On the other hand, standard approaches are usually more flexible in fitting data at both low and high workload intensities, and hence are able to produce more accurate interpolation results.

As an illustrative example, consider the TPC WS we used earlier. We provided every other data point collected during performance testing as training data, and the remaining data points were used to compute interpolation errors. We show results using the QN model and NN, because these results are most accurate among queueing models and standard regression approaches, respectively. Note that we present the results here as a motivation for the hybrid approach in Chapter 4.1.2; we will present a more comprehensive validation with other aforementioned models and WSs in Chapter 4.2. The results are depicted in Table 4.1.

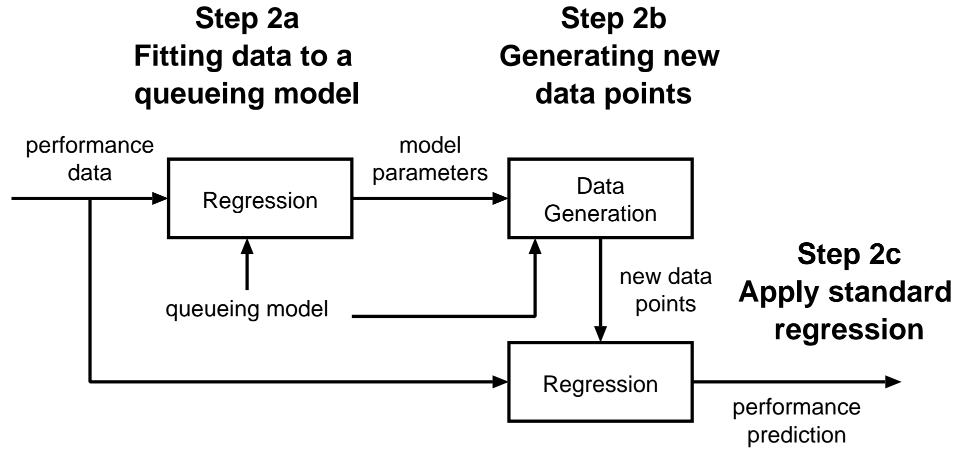


Figure 4.6: An overview of the hybrid approach

We can see that the interpolation errors of QN are higher than those of NN, e.g., when $\lambda_E = 0.7$, the error of NN is 0.01292 (or 1.136%), while the error of QN is 0.10358 (or 9.11%). These results have motivated us to derive a hybrid approach, that takes advantage of the low interpolation errors of standard regression approaches at low workload intensity, and more accurate extrapolation results of the queueing models at high workload intensity.

A Hybrid Approach

How do we take advantage of the better interpolation accuracy of standard regression approaches at low workload intensity, and the better extrapolation results of the queueing models at high workload intensity? Figure 4.6 illustrates our proposed hybrid approach. Recall that $\hat{\lambda}_E^{max}$ is the highest arrival rate sampled during performance testing.

The main idea is to first fit queueing models with performance testing data at the sampled arrival rates ($\lambda_E \leq \hat{\lambda}_E^{max}$, Step 2a), and then generate new performance data points at higher arrival rates ($\lambda_E > \hat{\lambda}_E^{max}$) using the fitted queueing model (Step 2b). In the final Step 2c, we augment the real performance testing data with the QN-predicted

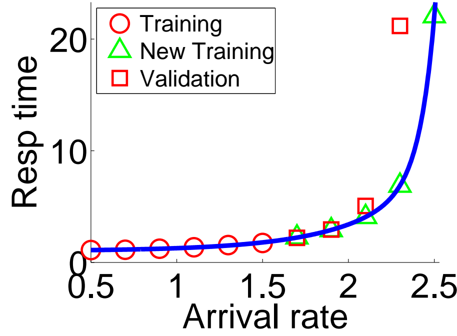


Figure 4.7: Results using QN^3

performance testing data. We then apply standard regression approaches to the augmented data to build a new prediction model which fuses knowledge from the queueing model.

We hypothesize that the resulting model has low interpolation errors at low workload intensity, as compared to using queueing models alone, while being able to extrapolate response time at high workload intensity, as compared to using standard regression approaches alone. The following example supports the hypothesis. More detailed validation results are given in the next section.

As an illustrative example, consider applying this hybrid approach to the TPC WS. Since the interpolation results using NN are most accurate among standard approaches, and the extrapolation results using QN are most accurate among queueing models (Figure 4.4(b)), we use QN in Steps 2a and 2b, and NN in Step 2c in the results to be presented here and in Chapter 4.2. We refer to this approach as QN^3.

Step 2a: We fit data collected during performance testing at the sampled arrival rates, depicted as circles in Figure 4.7, using a QN with 2 queues (introduced in Chapter 4.1.2). In this example, the parameters of the QN, obtained using regression analysis by supplying Eq. (4.17) as the regression function, are $\alpha_1 = 2.5908$ and $\alpha_2 = 2.5912$.

Table 4.2: Errors in response time estimates using QN^3

λ_E	measured	QN^3	QN	NN
0.7	1.13700	0.01381	0.07936	0.00310
0.9	1.22280	0.00072	0.04008	0.00536
1.1	1.35670	0.01968	0.01533	0
1.3	1.55030	0.03610	0.00113	0
1.5	1.74110	0.04974	0.09206	0
1.7	2.20000	0.02348	0.04464	0.32503
1.9	2.94880	0.04137	0.05449	1.00066
2.1	5.05620	0.98818	0.98297	3.07344
2.3	21.17940	14.29826	14.30680	19.18136

Step 2b: The next step is to generate new data points using this QN model by plugging in $\lambda_E > \hat{\lambda}_E^{max}$, α_1 and α_2 into Eq. (4.17). In our example, the new data points are depicted as triangles in Figure 4.7.

Step 2c: Finally, we take the data from Steps 2a and 2b as inputs to a standard regression approach (in our example NN), with results depicted in Table 4.2 and Figure 4.7.

The results in Table 4.2 indicate that the interpolation errors of QN^3 are comparable to using NN alone and are lower than using QN alone. At the same time, the extrapolation errors of QN^3 are very close to using QN, and are lower than using NN alone (which produces poor extrapolation results). These results illustrate that QN^3 is more accurate than using either QN or NN alone. A more comprehensive validation is presented next.

4.2 Validation

We perform an extensive evaluation and comparison of the approaches described in Chapter 4.1, i.e., standard regression techniques, queueing models (QN, M/M/1,

M/M/m, and M/G/1), and QN^3. Concretely, we analyzed 4 WSs with different configurations. We predict response times using above stated approaches and report their errors.

The 4 WSs which have analyzed are the AB WS and the TPC WS that we deployed in a controlled environment (both described earlier), and the Weather WS [7] and the Geocoding WS [1] that are "live". Analysis on other "live" WSs and "fictitious" WSs yielded similar conclusions and are thus omitted for brevity.

We report RMSE – (squared) root of measure squared errors – a commonly used evaluation metric in regression analysis. The errors are defined as the differences between the predicted values and the measurements (ground-truth). For each WS, we sent 10000 requests at a fixed arrival rate according to a Poisson process, and computed the average response time. This process was repeated at 9 - 11 different arrival rate values. The data was then split into two sets (with details given below): data in the training set was supplied as input to each approach, and we computed the approach's RMSE using its predictions and data in the validation set.

In what follows, we report first results on *interpolation*. In this setting, parameters of our models are estimated on training data (i.e., different arrival rates) whose value ranges are the same as validation data. Then, we report results on *extrapolation*, where the ranges of training data and validation data are disjoint. Our evaluation results show that, while other techniques perform well on either interpolation or extrapolation, QN^3 performs the best in *both* cases.

4.2.1 Interpolation Errors

In this set of experiments, we choose an odd number of data points. An example is the data in the first two columns in Table 4.2. We sort them according to the corresponding

Table 4.3: TPC WS interpolation errors

λ	measured	QN	M/M/1	M/M/m	M/G/1	Poly
0.7	1.13700	0.10358	0.52988	0.20194	0.45465	0.10844
1.1	1.35670	0.04127	0.55485	0.26512	0.39676	0.10039
1.5	1.74110	0.08451	0.56063	0.30143	0.24797	0.26089
1.9	2.94880	0.14917	0.71227	0.47936	**0.01250**	0.70997
		Exp	Splines	NN	GP	QN^3
		1.13034	0.16458	**0.01292**	0.08168	**0.01292**
		1.30735	0.16070	**0.02052**	0.04575	**0.02052**
		1.37532	0.55890	**0.04117**	0.08720	**0.04117**
		0.23740	1.77780	0.03971	0.65222	0.03971

Table 4.4: Average Interpolation Errors

	QN	M/M/1	M/M/m	M/G/1	Poly
TPC	0.0946	0.5864	0.3120	0.2780	0.2949
AB	1.7508	2.3515	2.3141	1.0578	0.4948
Geocoding	0.1847	0.2154	0.2228	0.2417	0.0876
Weather	**0.0430**	0.2340	0.0939	0.1308	0.0846
	Exp	Spline	NN	GP	QN^3
TPC	1.1026	0.6655	**0.0286**	0.2167	**0.0286**
AB	1.9163	0.5784	**0.2451**	1.2404	**0.2451**
Geocoding	0.1168	0.0787	**0.0513**	0.0659	**0.0513**
Weather	0.3587	0.1107	0.0878	0.3199	0.0878

arrival rates and then select the data points, alternating between the training and the validation data sets.

Note that since the first and the last data points are always selected for training data, we are guaranteed that the arrival rates in the validation data are always within the range of the rates in the training data.

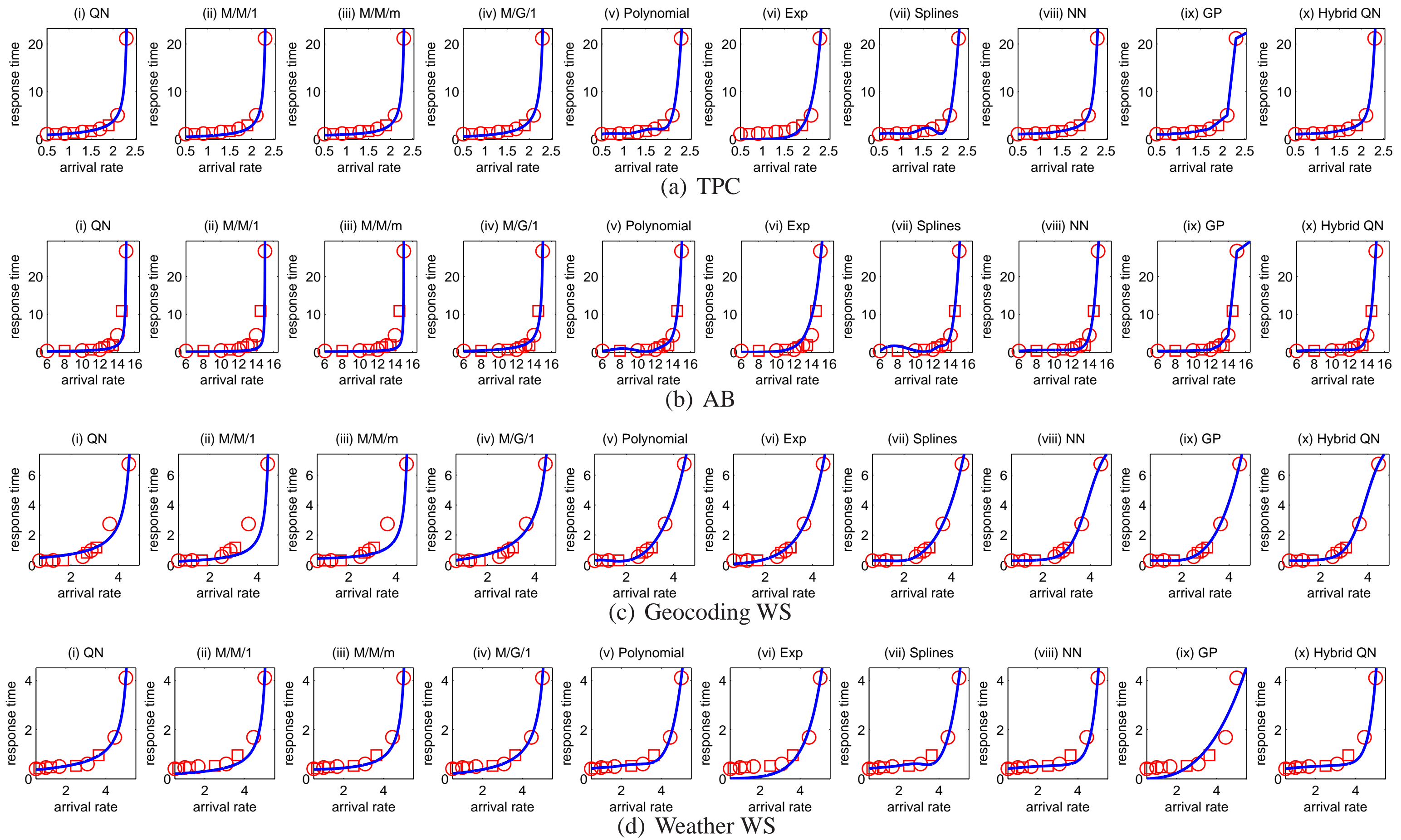


Figure 4.8: Interpolation

In Figure 4.8, we illustrate the fitted regression curves (draw in blue) along with the training data (using squares) and the validation data (using circles). In Table 4.3, we report the errors of the TPC WS at different arrival rates, and in Table 4.4, we report the average interpolation errors across all arrival rates for each of the 4 WSs - best performing techniques are shown in bold. Detailed results for the other 3 WSs have similar patterns to those reported in Table 4.3 and are thus omitted.

From Table 4.3, we observe that the M/M/1 and M/M/m models give higher interpolation errors than the QN model in the TPC WS. This illustrates that the QN model is a better description of the TPC WS than the M/M/1 and M/M/m models, because the TPC WS was deployed on four physical servers, and hence the QN model, which assumes a multi-queue system, describes the TPC WS more accurately than the single-queue systems.

From Table 4.4, we observe that while applying the QN model to the TPC, Geocoding, and Weather WSs had lower interpolation errors, it had higher interpolation errors than the M/G/1 model when it was applied to the AB WS. This is because (1) the AB WS was deployed on a single machine, in which the M/G/1 model had accurately described as a single-queue system; and (2) the QN model uses exponential service times, which is unlikely the case in our performance testing. The M/G/1 model, on the other hand, is able to more accurately capture the service time distribution, as it assumes a general service time distribution. This illustrates that the M/G/1 model is more accurate if the WS is a single-server WS. Since we do not know if a WS being tested is a single-server or multi-server system, these results indicate that we should use a combination of queueing models, because none of the queueing models outperformed the others.

Now let us study the accuracy of using polynomials. We experiment with polynomials of different degrees to fit the results of the 4 WSs, and present the results with the lowest interpolation errors in Figure 4.8: an 8th-degree polynomial for the TPC WS,

a 12th-degree polynomial for the AB WS, a 4th-degree polynomial for the Geocoding WS, and a 3rd-degree polynomial for the Weather WS. From Table 4.4, the interpolation errors of using polynomials are similar to the queueing models, and outperform all four queueing models in the Geocoding WS. We conclude that the use of polynomials gives similar interpolation results as the queueing models.

While the exponential model gives good predictions when the arrival rate is high, the predictions are lower than the measured response time when the arrival rate is low, which results in high errors. This is more visible in the TPC WS (Figure 4.8(a)(vi)), in which the model has underestimated the response time when $\lambda_E < 1.5$. This is because the exponential function increases at a different rate than the measured data. In fitting the data, the regression algorithm "sacrifices" the accuracy of the response time at low workload intensity. The reason is that if the regression algorithm fits the data at low workload intensity, the rate that the response time goes up would be too low in the exponential model, and hence causing large errors when the arrival rate is high. If the regression algorithm fits the data at low arrival rates, the large errors at high workload intensity offset the small errors at low workload intensity. Therefore, the regression algorithm chooses to sacrifice the accuracy at low workload intensity. For this reason, we concluded the exponential function is not a good function to model WS response time, as it underestimated the response time when the WS was lightly-loaded.

In our experiments, splines exhibited overfitting, which is characterized by decreases in response times even when the arrival rates increase (e.g., in Figure 4.8(b)(vii)). This undesirable property makes it not a good approach for interpolation.

In general, from Table 4.4, NN and GP had lower interpolation errors than the queueing models, and NN had lower errors than GP. For example, in the Geocoding WS, the interpolation errors of NN and GP (0.0513 and 0.0659, respectively) were lower than the most accurate queueing model (QN, whose error is 0.1847). However, we observed

that the interpolation errors of GP were noticeably higher than those of the queueing models in the TPC WS. This is because GP used a straight line to connect data points at high workload intensities, causing high interpolation errors when $\lambda_E = 2.1$ in Figure 4.8(a)(ix). Despite the possibility of overestimation at high workload intensities, the results have indicated that NN and GP are better approaches than using queueing models for interpolation. We consider accuracy in interpolation an advantage of standard regression approaches over queueing models.

Note that the results of QN^3 were the same as NN in this experiment. This can be explained as follows: since we supplied data at high workload intensities (i.e., $\lambda_E \simeq \hat{\lambda}_E^{max}$), little or no new data is generated in Step 2b. Hence, the data supplied to NN in QN^3 in Step 2c was the same as the data supplied to NN when it was to be used alone.

4.2.2 Extrapolation Errors

The next experiment studies how well the models predict response times beyond the range of arrival rates used in performance testing. As in the results presented in Chapters 4.1.2 and 4.1.2, the training set consists of data points corresponding to arrival rates in the lower 60%, and the evaluation set consisted of data points corresponding to arrival rates in the upper 40%.

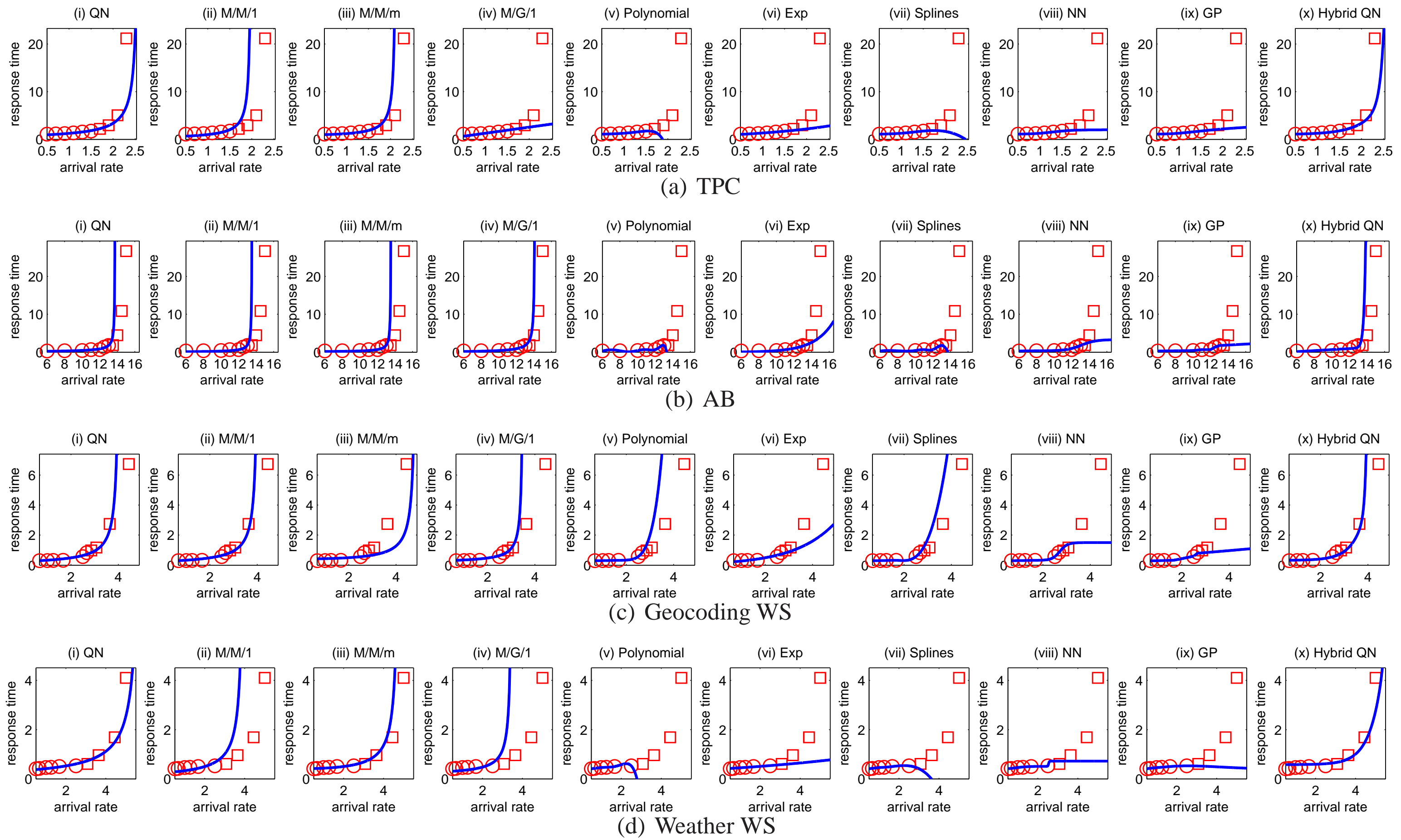
The results are depicted in Figure 4.9 and Table 4.5. As in the results in Chapter 4.1.2, the standard regression approaches predicted increases in response time at much slower rates in many cases. For example, the standard regression approaches predicted the response time staying flat, except in the Geocoding WS, in which polynomial and spline correctly predicted the response time increasing (Figures 4.9(c)(v) and 4.9(c)(vii)). This is because the response time had started to increase rapidly when $\lambda_E = 2.3$. Polynomial and spline even predicted the response time to go down in the TPC, AB, and Weather WSs. This provides evidence that standard regression

Table 4.5: Extrapolation Errors

	$\hat{\lambda}$	measured	QN	M/M/m	QN^3
TPC	1.70000	2.20000	0.04464	0.39427	**0.02348**
	1.90000	2.94880	0.05449	1.67352	**0.04137**
	2.10000	5.05620	**0.98297**	30.12275	0.98818
	2.30000	21.17940	14.30680	-	14.29826
AB	13.50000	1.85298	**2.66497**	17.90088	3.87008
	14.00000	4.46033	-	-	58.95611
	14.50000	10.85322	-	-	676.94406
	15.00000	26.64767	-	-	2495.16183
Geocoding	2.85710	0.97300	0.14335	0.34484	**0.08964**
	3.07690	1.17120	**0.15661**	0.48525	0.16062
	3.63640	2.74360	**0.39774**	1.81043	0.41562
	4.44440	6.72810	-	4.22958	277.91401
Weather	3.07690	0.60660	0.13125	0.10086	**0.06777**
	3.63640	0.96270	**0.03293**	0.01556	0.14722
	4.44440	1.68640	**0.19728**	1.23738	0.22925
	5.00000	4.09690	1.55715	-	**1.35149**

approaches are not effective at extrapolating models in terms of handle inputs outside of the range of their training data. As discussed earlier, this is a major shortcoming, because it is often infeasible to do performance testing at high workload intensities, as discussed in Chapter 4.1.1. However, in order for these approaches to accurately predict response time at high workload intensities, they require data at high workload intensities, which can overload the system being tested.

The queueing models performed better than the standard regression approaches, as they predicted the rapid increase in the response time, when arrival rates were high. We observed that while the M/M/1 and M/G/1 models correctly predicted the rapid increase in response time, they were more pessimistic than the QN model. This is because they assume a single-server, whereas WSs are typically not. Thus, the two single-server models overestimated the utilization of the system, and hence gave pessimistic results.



(a) TPC

(b) AB

(c) Geocoding WS

(d) Weather WS

Figure 4.9: Extrapolation

In addition, the M/G/1 model was unable to predict response time in the TPC WS at high workload intensity (Figure 4.9(a)(iv)). Upon closer examination of the estimated parameters, in this particular example, the regression algorithm estimated the service rate to be very high ($\mu \simeq 4000$), which was much larger than other queueing models (e.g., $\mu = 2.31$ in the M/M/1 model). This indicates that the flexibility in the service time distribution of the M/G/1 model may cause poor extrapolation results, and therefore the M/G/1 model should be used along with other queueing models in extrapolation.

Qualitatively, the results of the M/M/m model were comparable to those of the QN model: the results were similar in the AB WS, while the M/M/m model was more optimistic in the Geocoding WS, and it was more pessimistic in the TPC and Weather WSs. To compare the two models more closely, we tabulate the extrapolation errors in Table 4.5. An "–" in the table indicates that the model predicts the system as being unstable at that arrival rate. As we can see from the table, the QN model had lower extrapolation errors than the M/M/m model in all WSs, except for the Geocoding WS, in which the QN model was more pessimistic, and considered the system as unstable when $\lambda_E = 4.44$. This indicates that the QN model is a better model than the M/M/1, M/G/1, and M/M/m models.

The extrapolation results of QN^3 were comparable to the results of QN, as we used QN for extrapolation. These results indicate that QN^3 has lower extrapolation errors than NN (which is unable to extrapolate), and that the extrapolation results of QN^3 are comparable to those of using QN alone.

Summary: Combining our results in Tables 4.4 and 4.5, we observe that QN^3 can perform well at both, interpolation and extrapolation tasks and is better than using standard regression approaches or queueing models alone.

4.3 Conclusion

It has become more common to integrate third-party software components for creation of new systems; hence it is important to understand performance characteristics of third-party components. To reduce the cost of performance testing, we estimate the performance of third-party components during high workloads using data collected at low workloads, and apply our approach to performance estimation of WSs. Our hybrid approach combines the low interpolation errors of standard regression analysis with the low extrapolation errors of queueing models for response time prediction. Our validation results indicate that the hybrid technique is accurate, as compared to using standard regression approaches or queueing models alone. Thus, we believe that our technique can be used to improve system that involve third-party components. For instance, our approach can be utilized by service selection techniques [16, 48]. In this context, a WS can be composed dynamically, where performance characteristics can be part of the selection criteria. Our approach can support such techniques by providing performance estimation information for a given WS, i.e., so that such approaches can make more informed decisions.

Chapter 5

Parameter Estimation in Quality Analysis of Software Components

As we discussed in Chapters 1 and 2, a major obstacle in design-time software quality analysis techniques is that it is difficult to reliably determine a software system's operational profile, because the implementation is not available. Existing approaches simply assume the operational profile, which describes the system's or component's usage, is available, and have not adequately addressed this problem.

In this chapter, we focus on operational profile estimation in component reliability prediction. Estimating the operational profile for performance prediction requires integrating information about the performance of underlying firmware (e.g., operation systems, middleware, and hardware) into the analysis, which is part of our future work (see Chapter 6.2.1).

In Chapter 5.1, we describe a component-level reliability prediction framework [19], and highlight the parameters it requires. In Chapter 5.2, we describe sources of information that are available during the design stage, and describe how they can be used in generating operational profiles. Finally, in Chapter 5.3, we compare our results with results obtained from an implementation, which are used as "ground truth". While we focus on operational profile estimation at the component level, we believe what we propose in this chapter also applies at the system level.

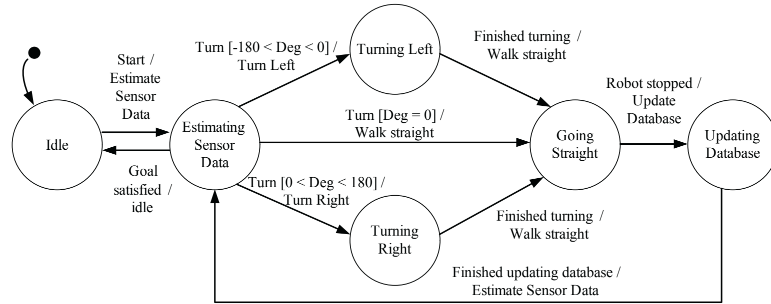


Figure 5.1: Dynamic Behavior Model of the *Controller* Component

5.1 Component Reliability Prediction Framework

Before we describe our operational profile modeling process, we first describe a component that we use as a running example throughout this chapter. The example that we use in this chapter is that of the *Controller* component of the SCRover (depicted in Figure 5.1), a third-party robotic testbed based on NASA JPL's Mission Data System framework [15]. 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.

Here, we describe a component reliability prediction framework in [19] that we apply our operational profile estimation approach to. For ease of exposition, we present this framework as a three-phase process depicted in Figure 5.2.

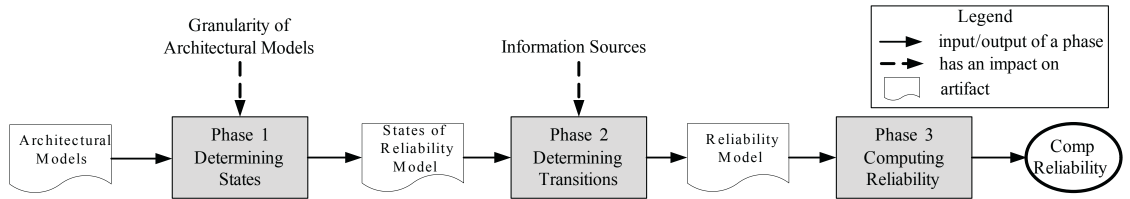


Figure 5.2: Software Component Reliability Prediction Framework

In Phase 1, we determine an appropriate set of states S of the component's reliability model by leveraging architectural models. In Phase 2, we determine the values of the transition matrix P of the reliability model by leveraging information available at the architectural level. Finally, in Phase 3, we compute component reliability by applying standard techniques [70]. We briefly describe each step in the remainder of this section.

We focus on Phase 2 in this chapter. This involves estimating an operational profile of a component, which is represented by transition probabilities in the reliability model. Details of Phase 2 is given in Chapter 5.2.

Phase 1: Determining States

In Phase 1 of the component reliability prediction framework, we determine the set S by leveraging architectural models and performing standard architectural analyses [47]. There are two types of states in the set S that need to be determined: states corresponding to component's normal behavior, B, and to faulty behavior, F.

We leverage a component's dynamic behavior model [61] 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 5.1 and consists of six states: *Idle* (B_1),

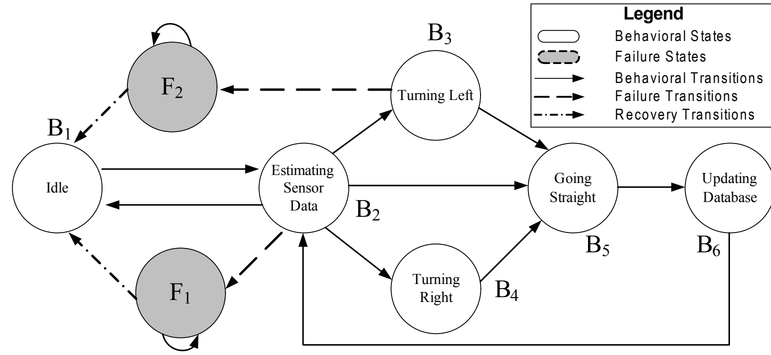


Figure 5.3: Reliability Model of the *Controller* Component

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 5.3).

To determine the failure states (set F) we analyze the architectural models of a component. The multi-view approach to modeling a component described in [61] allows for the detection of architectural inconsistencies. Standard techniques for architectural analysis [47] 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. Once we have identified the defects, we designate a failure state for each class of defect. For example, we identified two defects in the *Controller* component in Figure 5.1: Defect d_1 affects the *Estimating Sensor Data* state, and Defect d_2 affects the *Turning Left* state. We model the two defects as different classes, and designate two failure states $F = \{F_1, F_2\}$ to correspond to the Defects d_1 and d_2 respectively.

Phase 2: Determining Transitions

Values of the transition matrix P are determined in this phase using various sources of available information. Given the states, determination of transition probabilities

between these states remains a challenge. A critical difficulty here is the lack of information about the operational profile and failure information of the component. We address this problem by (a) identifying and classifying the utility of information sources available during architectural design and (b) combining the use of such sources with a hidden Markov model (HMM)-based approach that was proposed in [60]. The description of information sources typically available at the architecture level and the details of determining transition probabilities are described in Chapter 5.2.

Phase 3: Computing Reliability

Once the states and the transition probabilities of the Markov chain reliability model are determined, in Phase 3 of the component reliability prediction framework, the model is solved to compute a reliability prediction.

Let $\pi(i)(t)$ be the probability that a component is in state i at time t, where $i \in B \bigcup F$. As t goes to infinity (i.e., as the component operates for a long time), these probabilities converge to a stationary distribution [69],

$$\vec{\pi} = [\pi(F_1), \ldots, \pi(F_M), \pi(B_1), \ldots, \pi(B_N)] \tag{5.1}$$

which is uniquely determined by the following equations [69]: [1]

$$\begin{aligned} \textstyle\sum_{i \in S} \pi(i) = 1 \\ \vec{\pi} = \vec{\pi} P \end{aligned} \tag{5.2}$$

This system of linear equations can be solved using standard numerical techniques [70]. The component's reliability can then be defined as the probability of not being in a failure state:

[1]It is not difficult to show that for our reliability model this limiting distribution exists and is a stationary one [69].

$$R = 1 - \sum_{i=1}^{M} \pi(F_i) \tag{5.3}$$

As an illustrative example, let the transition matrix P, estimated using the approach to be described in Chapter 5.2, is as follows:

$$\begin{array}{c} F_1 \\ F_2 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \end{array} \begin{pmatrix} 0.8 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0.05 & 0 & 0.019 & 0 & 0.076 & 0.0095 & 0.8455 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0 & 0.96 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{5.4}$$

After solving Equation (5.2), we have

$$\begin{aligned} \vec{\pi} &= [\pi(F_1), \pi(F_2), \pi(S_1), \pi(S_2), \pi(S_3), \pi(S_4), \pi(S_5), \pi(S_6)] \\ &= [0.0765, 0.0012, 0.0220, 0.3061, 0.0233, 0.0029, 0.2840, 0.2840] \end{aligned}$$

Thus, the reliability of the *Controller* component is

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

5.2 Operational Profile Modeling

Estimating the transitions in the reliability model involves estimating an operational profile of a component. The transitions in our reliability model, corresponding to the elements of the transition probability matrix P, can be viewed as being of three different types: (1) behavioral; (2) failure; and (3) recovery. Behavioral transitions are between

two behavioral states; failure transitions are from a behavioral state to a failure state; and recovery transitions are from a failure state to a behavioral state. The process of determining the probabilities of each transition may be different and depends on the information available to the architect.

We identify the following 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.

  For example, consider estimating the outgoing transition of the *Estimating Sensor Data* state in the *Controller* component in Figure 5.3. To determine the transitions to *Turning Left*, *Turning Right*, and *Going Straight* states, we can ask the expert to estimate the probability that a robot turns. If the expert predicts the robot to be going straight most of the time, we can estimate the transition probability from *Estimating Sensor Data* to *Going Straight* to be larger than the transition probability going to the *Turning Left* and *Turning Right* states.

- **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.

For example, in SCRover's requirements document [4], one of the requirements states the acceptable time to reboot SCRover in case of a software crash. This information can be used in estimating the recovery probabilities.

- **Simulation**. Simulation of a component's architectural models [31] 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.

  For example, relying on the domain expert on estimating the parameters of a complex components may be error-prone, because of the complexity of the component. The domain expert, on the other hand, may be able to suggest possible operational profiles at a higher-level, in which each higher-level event may correspond to multiple transitions in the component's simulation model. We explore this technique in our evaluation in Chapter 5.3.1.

- **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. A strength of our approach is that we can use them in a complementary manner in order to 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 5.1, 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.

Our work in [60] suggests that in such a case we can use hidden Markov models (HMMs) [55] to obtain behavioral transition probabilities. An HMM is defined by a set of states $S = \{S_1, S_2, \ldots, S_N\}$, a transition matrix $A = \{a_{ij}\}$ representing the probabilities of transitions between states, a set of observations $O = \{O_1, O_2, \ldots, O_M\}$, and an observation probability matrix $E = \{e_{ik}\}$, which represents the probability of observing event O_k in state S_i. The set S of the HMM comes from Phase 1. The

event/action pairs of the dynamic behavior model become observations of the HMM (set O).

Once we have determined the set of states S and observations O of the HMM, we can apply the Baum-Welch algorithm [55] to estimate the transition probabilities. The inputs to the algorithm are (1) a starting point, corresponding to initializing the Matrices A and E, and (2) training data for parameter estimation. The Matrices A and E can be initialized with random values [55] or they may be initialized more intelligently, by utilizing architectural models. Since the Baum-Welch algorithm is a local optimization technique, the starting point (given by the Matrices A and E) can affect the accuracy of the output. Therefore, to obtain an accurate operational profile, it is important to start at a "good" starting point. We observe that, typically, it is unlikely that all event/action pairs can happen in all states. Thus, it is possible to determine which entries in the Matrix E are zero (i.e., $e_{ik} = 0$ when event k cannot happen in state S_i), and fill in random values for other possible event/action pairs. The information on possible event/action pairs at the states is available from the component's architectural models. For example, in the *Controller* component in Figure 5.1, the *Start* event is not possible when the component is in state *Turning Left*. Therefore, we can set the corresponding entry in the Matrix E to 0.

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 does not include any failure or 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. 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: $f_{ij} > 0$ if Defect d_j may occur in state B_i. This would be determined as part of the architectural analysis process, as described in Phase 1. Also, for 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_l, we can determine whether r_{kl} is non-zero: $r_{kl} > 0$ if the component restarts in State B_l after recovery from a failure caused by Defect d_k. In the *Controller* component from Figure 5.3 defects of classes D_1 and D_2 can occur in states B_2 and B_3, respectively. Thus, we add transitions (with non-zero probabilities) from B_2 to F_1, and from B_3 to F_2. In this example, recovery from any failure returns the component back

[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 may not be reasonable in the case of multi-threaded components. We treat such complex components as systems and apply our system-level reliability prediction technique in Chapter 3 on them.

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. Estimating such failure-related information is challenging, because software engineers most often design components for correct behavior, information related to failure are limited. 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 Chapter 5.3. 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 help the reliability modeler determine how difficult it is to recover from a failure due to defect class D_k. In turn, this would indicate the values ranges for r_{kl} the reliability modeler should consider.

5.3 Evaluation of Operational Profile Estimation

In this section, we validate and support several claims we have made throughout this chapter. 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 Chapter 5.2. Since our approach is intended to be used at design-time, a direct comparison of reliability numbers predicted

by the approach and those measured at runtime would not be meaningful.[3] 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 Chapter 5.2). 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 evaluated our approach 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 [60];
- the SCRover robotic testbed [15], developed by a separate research group at USC in collaboration with NASA's JPL;
- MIDAS [45], a large, embedded system developed as part of a separate collaboration between USC and Bosch;
- DeSi [44], 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 over the past decade [9].

[3]For example, at implementation time, it may be appropriate to evaluate a system's reliability using the five 9's standard. However, this is not typically meaningful at design time.

In order to observe the trends in our approach'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 approach has consistently yielded qualitatively similar results for all of the above cases. To illustrate these results and highlight the approach's key properties, particularly its sensitivity, we will use SCRover's *Controller* component in Chapter 5.3.1, as well as a component from the DeSi environment [44] in Chapter 5.3.2. Results from a number of other components we have evaluated are qualitatively similar, and they are available in [2].

5.3.1 Evaluation of SCRover's Controller

In this section, we present sensitivity analysis of the *Controller* component we used as an example throughout this chapter. We study the sensitivity of our results to (a) different information sources identified in Chapter 5.2, (b) changes to operational profile, and (c) models of different granularities. To validate our results, we constructed a detailed behavioral (control-flow) model of the *Controller* from a prototype implementation of the component that had existed previously. This implementation-level model is based on a directed graph that represents the component control structure. We then built a Markov model by leveraging this graph, where a node in the graph translates to a state in the Markov model. This is analogously to what existing approaches have done at the system level (e.g., [20, 35]). Based on the available component maintenance records, we injected defects into the code to simulate failure behavior. We should note that we were not interested in implementation-level faults in this model (e.g., an implementation-level defect that may cause a division by zero error), but only in architectural defects. To ensure a fair comparison with our architectural-level model, we assume there is no

implementation-level defects, as these defects are not modeled at the architectural level. We then introduced failure states and transitions in the control structure to represent erroneous behavior corresponding to the injected defects. The results obtained from this model were used as "ground truth" in a large number of experiments.

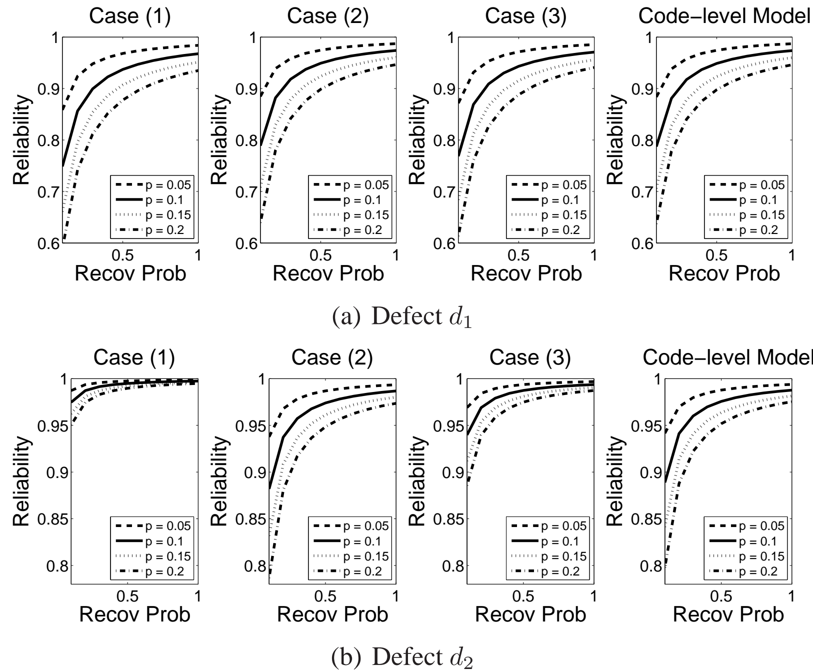
As described in Chapter 5.2, our approach allows for multiple failure classes. However, for clarity of exposition of results, in what follows, experiments are performed using one active class of defect at a time. In the presented experiments, this is done by setting probabilities of failures associated 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 [2].

Sensitivity to Information Sources

We study the sensitivity of our approach to different information sources. The following sources of component usage information were considered in this evaluation. Here, we present the parameter values we used in generating Figure 5.4. We have performed similar analyses with different inputs, and we obtained qualitatively similar results.

- Case (1) - Domain Expert - We were given the architectural models, and focus on operational profiles that the expert suggests.

- Case (2) - Simulation - We were provided with SCRover's requirements, based on which we specified a sequence of high-level events to simulate the dynamic behavior model of *Controller* shown in Figure 5.1. We obtained training data by



(a) Defect d_1

(b) Defect d_2

Figure 5.4: Analysis of sensitivity to information sources of SCRover's *Controller*

leveraging the simulation trace and applied our HMM-based approach to obtain behavioral transition probabilities (recall Chapter 5.2).

- Case (3) - Functionally similar component - As a functionally similar component we selected a robot that walks from one point to another, and avoids obstacles along the way. We then used an operational profile of this component in our reliability prediction of the *Controller* component using our HMM-based approach described in Chapter 5.2.

In one set of experiments, we were interested in the sensitivity of the component reliability when the probabilities of recovering from defects change. To this end, we fixed the failure probability, and varied recovery probabilities from 0.2 to 1.0 in 0.2 increments. We repeated the experiments for different failure probabilities. In Figure 5.4(a) we introduced Defect d_1, affecting the reliability of the *Estimating Sensor Data* state,

and in Figure 5.4(b) we introduced a Defect d_2, affecting the reliability of the *Turning Left* state.

Not surprisingly, we observe that the trends conform to our expectations in all three cases: as recovery probability increases, the reliability of the component increases since the time taken to recover from a failure becomes shorter. Moreover, as failure probability increases, component reliability decreases.

We note that in Figure 5.4(b) (Defect d_2), the slope of the curves in Case (1), where we have domain knowledge, is different from other cases. The reason is that our expert incorrectly predicted the robot to be walking mostly straight: in the prototype the robot walked at an angle most of the time, such that occasionally it was too far from, or too close to, the wall, and had to turn. As a result, the robot spends less time in the *Turning Left* state of the model generated based on our expert' predictions for Case (1) than it does in the *Turning Left* state of the actual system. Hence, Defect d_2 had less impact on the component's reliability.

Sensitivity to Operational Profile

To evaluate our reliability approach'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 5.3), except for a specific set. By varying those remaining transition probabilities, we can observe the model's response. In this experiment, we consider the ranges of *Controller*'s reliability values when the probability of going from state *Estimating Sensor Data* to state *Turning Left* (recall Figure 5.1) varies between 0 and 0.85, and adjust the probability of going to the *Going Straight* state accordingly. We fix the probability of going from state *Estimating Sensor Data* to state *Turning Right* and to state *Idle* at 0.1 and 0.05, respectively. All

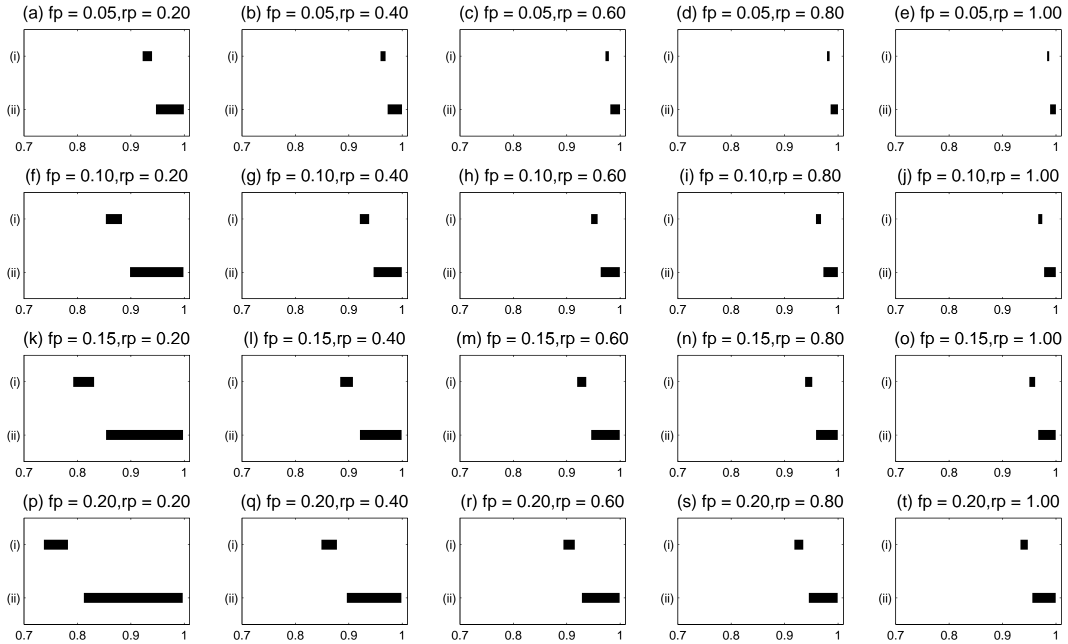


Figure 5.5: Analysis of sensitivity to operational profiles of SCRover's *Controller*

other parameters in the operational profile are fixed. This corresponds to estimating the probability that a robot turns left.

We reiterate that the same analysis was performed by varying transition probabilities between other states, and yielded qualitatively similar results. We varied the failure and recovery probabilities (as in Chapter 5.3.2), and obtained a reliability range for each failure-recovery probability pair. We did this for the two defects we introduced earlier.

Figure 5.5 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 *Estimating Sensor Data* to state *Turning Left* between 0 to 0.85. The bars labeled (i), and (ii) represent the Defects d_1 and d_2, 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 5.5. 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.

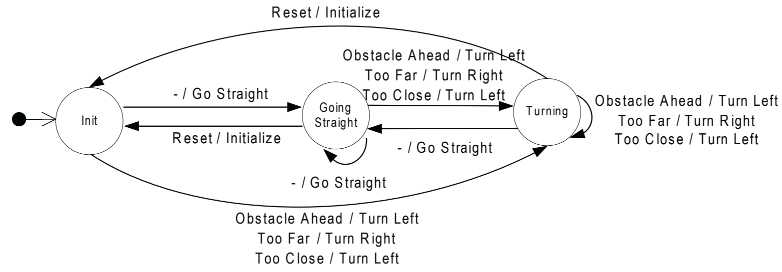
Another observation is that *Controller*'s reliability was more sensitive to Defect d_2. This is because d_2 directly affect the two states on which we focused in this particular scenario. More generally, by varying operational profiles, we can identify which defects most prominently affect the resulting reliability values across these operational profiles. If a defect is shown to increase the model's sensitivity to multiple operational profiles, software designers may want to focus their attention particularly on eliminating that defect in order to achieve the greatest improvement in the component's reliability.

Sensitivity to Model Granularity

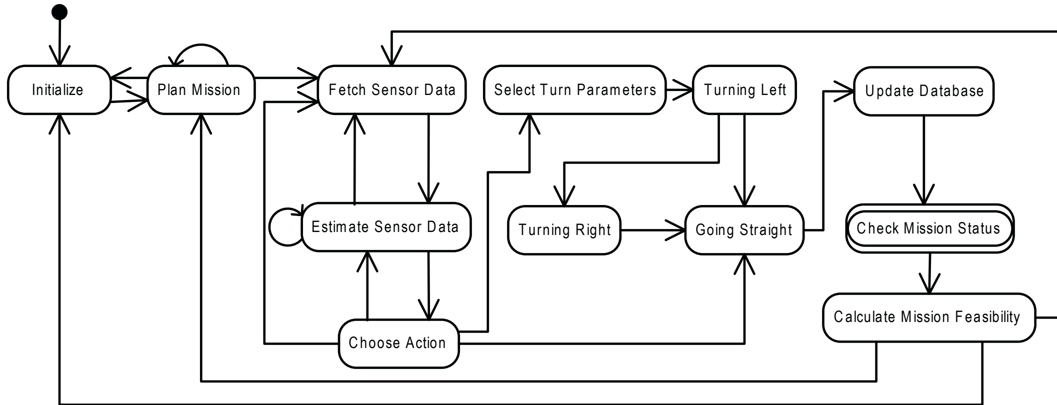
Software architectural models may vary widely in terms of the amount of detail they contain. Different models are produced at different points during the system's development, and may be intended for different stakeholders. On the average, it is possible to produce high-level models earlier than detailed 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 elaborate below, it 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 approach. To this end, we have performed sensitivity analyses on component models of varying granularity levels.

Figure 5.7 shows the results of such analysis using Defect d_1 for Case (1) discussed in Chapter 5.3.1. We obtained qualitatively similar results when we introduce Defect d_2,



(a) 3-state model



(b) 12-state model

Figure 5.6: Dynamic behavior models of the *Controller* component at two different levels of granularity.

or use different information sources. The six-state model is the example we have used throughout this paper (depicted in Figure 5.1). The three-state and twelve-state models are depicted in Figures 5.6(a) and 5.6(b), respectively. Note that the transition labels are omitted from Figure 5.6(b) for clarity.

We observe that in both cases, when recovery probability is fixed and failure probability increases from 0.05 to 0.2, reliability values are most sensitive in the three-state model. The other observation is that the three-state model is more sensitive than the six-state model to recovery probability, while the twelve-state model is least sensitive.

This trend can be explained as follows. Failures corresponding to Defect d_1 only occur in the *Turning Left* state in the twelve-state model. On the other hand, the time

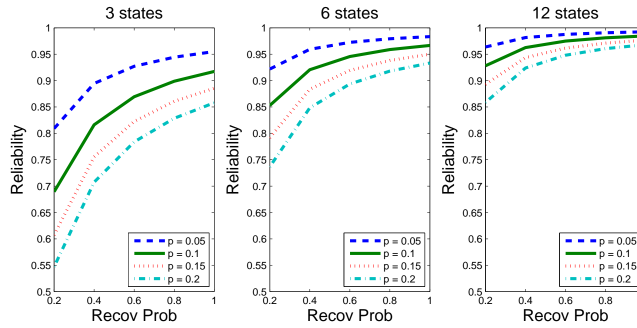


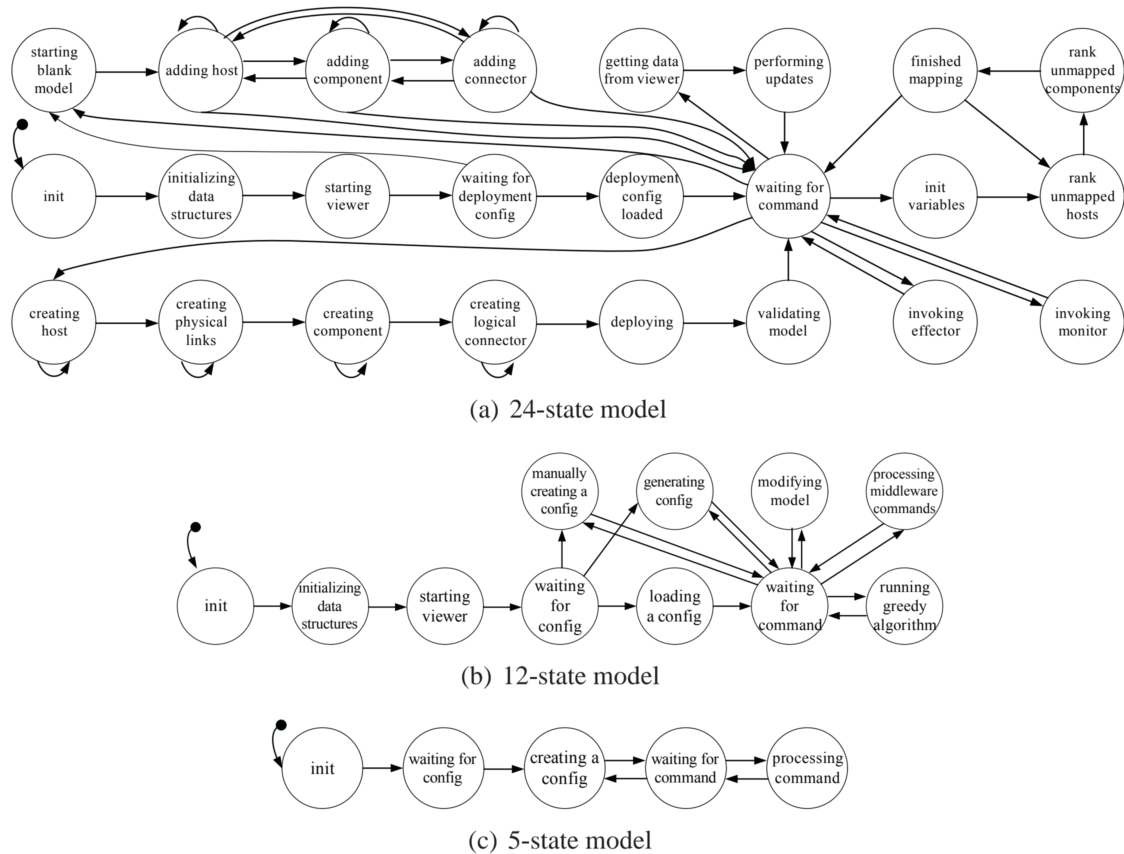
Figure 5.7: Analysis of sensitivity to models of different granularities of SCRover's *Controller*

spent in the *Turning Left* state in the six-state model also includes the time spent in the *Select Turn Parameters* state in the twelve-state model. As a result, the robot spends more time in the *Turning Left* state in the six-state model than in the twelve-state model, hence the sensitivity is higher in the six-state model. Analogously, since the time spent in the *Turning* state in the three-state models includes the time spent in *Estimating Sensor Data* and *Updating Database* states in the six-state model, the sensitivity of the three-state model is higher than that of the six-state model.

Note that in our experiments a model with fewer states gives more pessimistic results. We argue that, in general, it is (a) desirable for a approach such as ours to provide more conservative reliability 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 approach's predictive power.

5.3.2 Evaluation of DeSi

The SCRover's *Controller* component may be too small as a representative of real-world software components. Therefore, in order to study our approach more comprehensively,



(a) 24-state model

(b) 12-state model

(c) 5-state model

Figure 5.8: Architectural models of the *DeSiController* component at different levels of detail

we also perform sensitivity analysis on DeSi. DeSi is an environment that supports specification, manipulation, and visualization of deployment architectures for large distributed systems. It 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 [44] that update the *DeSiModel*. To demonstrate our approach's ability to handle components of large scale and complexity, we treat each subsystem as a single component.

Table 5.1: Defects injected in *DeSiController*

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

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 by existing architectural models. For consistency, we show the evaluation results of applying our approach to the *DeSiController* component only. A slightly abridged dynamic behavior model of *DeSiController* is depicted in Figure 5.8(a). To evaluate our approach in a controlled manner, we injected architectural defects into DeSi. Table 5.1 summarizes the subset of defects used in the results presented in the remainder of this chapter.

As in the evaluation of SCRover's *Controller*, to validate our results, we built separately a reliability model from the existing implementation of the *DeSiController* component, analogous to what we have done in Chapter 5.3.1. Again, we used the results obtained from this implementation-based model as the "ground truth" in our evaluations.

Sensitivity to Information Sources

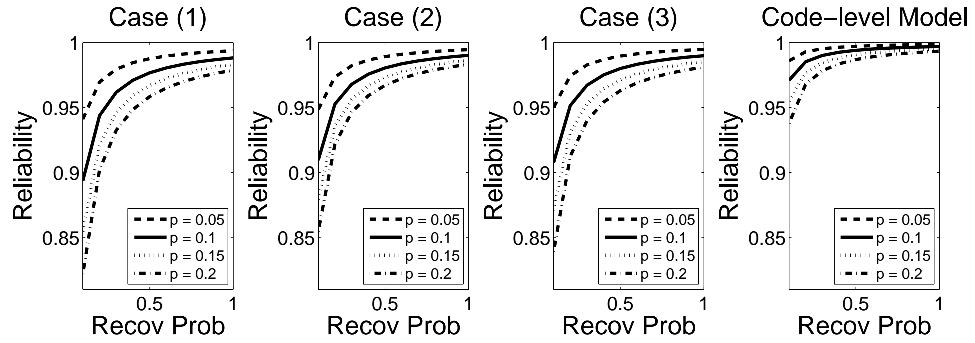
As in the evaluation using SCRover's *Controller* component in Chapter 5.3.1, we performed sensitivity analysis on models built using different information sources. We fixed the failure probabilities, and varied recovery probabilities from 0.1 to 1.0, at 0.1

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.

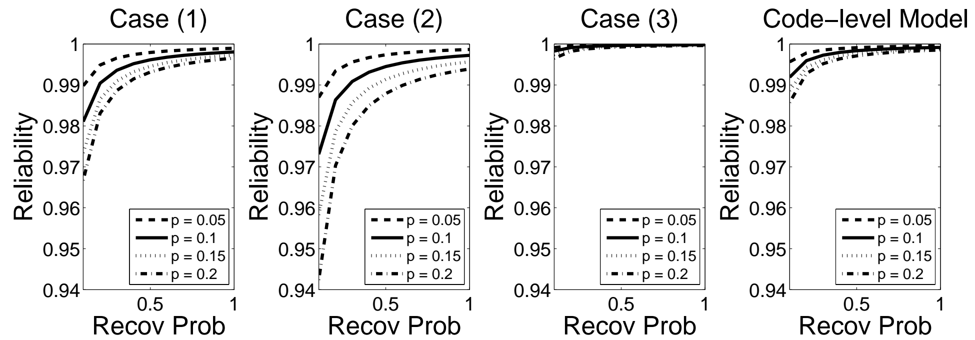
- Case (1) - Domain Expert - We relied on the information provided by DeSi's primary developer, and explored only the operational profiles suggested by him.
- Case (2) - Simulation - We were provided with DeSi's requirements [44], based on which we specified a sequence of high-level events to simulate the dynamic behavior model of *DeSiController* shown in Figure 5.8(a). We obtained training data by leveraging the simulation trace and applied our HMM-based approach to obtain behavioral transition probabilities (recall Chapter 5.2).
- 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 5.9, 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 5.1 in Figure 5.9(a), defect d_2 in Figure 5.9(b), defect d_3 in Figure 5.9(c), and defect d_4 in Figure 5.9(d). As in the case of SCRover's *Controller*, we observe that the trends conform to our expectations in all four cases for all defects.

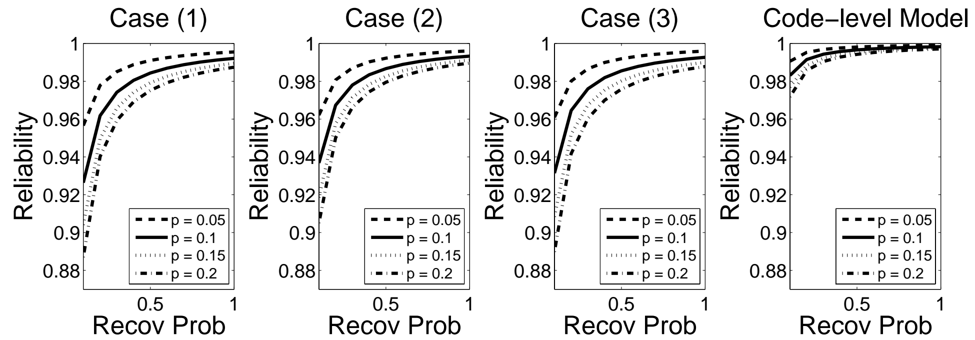
Although the general trends across the experiments are similar, Figure 5.9 yields some interesting observations. First, the sensitivity of the Case (1) results, and their accuracy as compared to the implementation-level model results, varies depending on the defect being studied. We have observed this situation in a number of other examples. As in the results in Chapter 5.3.1, this indicates that information provided by an expert



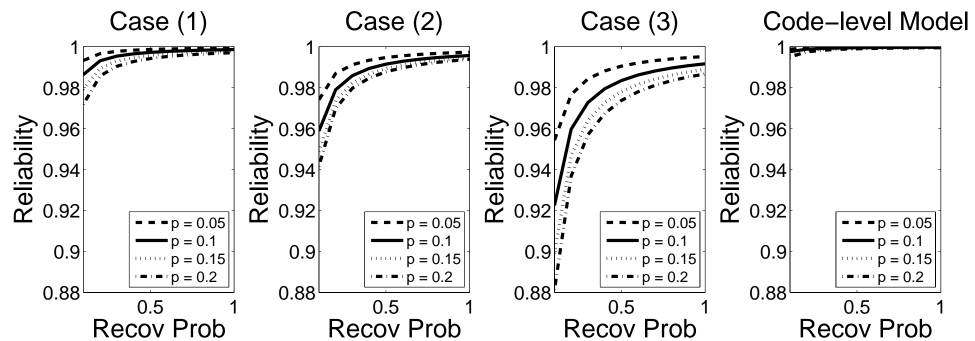
(a) Defect d_1



(b) Defect d_2



(c) Defect d_3



(d) Defect d_4

Figure 5.9: Analysis of sensitivity to information sources of *DeSiController*

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 5.9(b), 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 5.9(d), 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 5.9, the implementation-level model exhibits higher reliability than the other cases. This occurs because the implementation-level model is finer-grained than the architectural models. As we have shown in Chapter 5.3.1, coarser-grained models give more conservative results in our approach.

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).

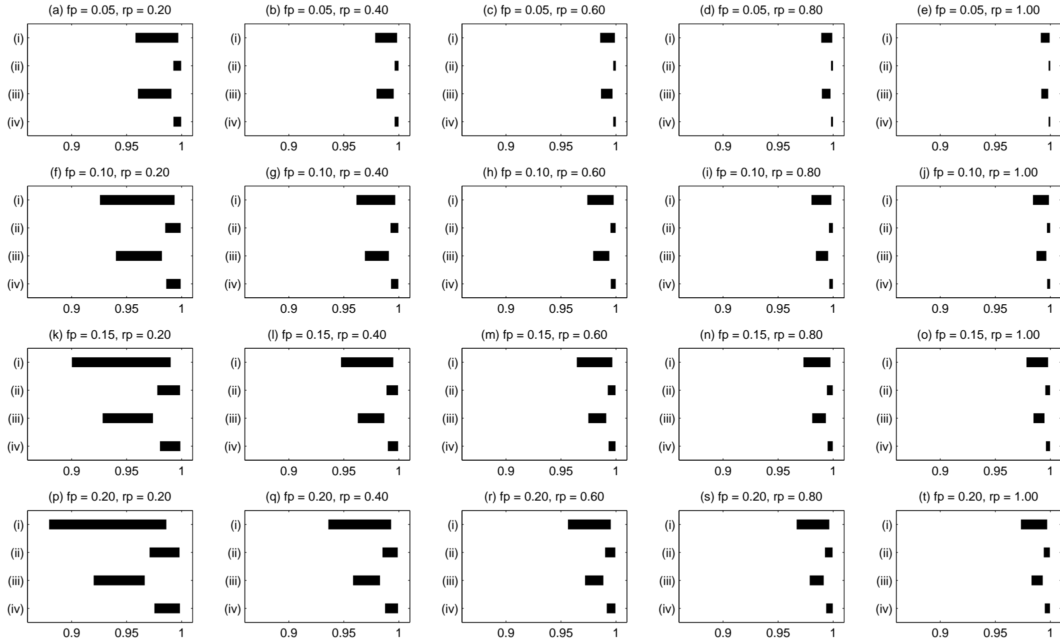


Figure 5.10: Analysis of sensitivity to operational profiles of *DeSiController*

Sensitivity to Operational Profile

We study the effect of changes in operational profiles to component reliability, similar to what we did in evaluating our approach using SCRover's *Controller*. We consider the ranges of *DeSiController*'s reliability values when the probability of going from state *Finished mapping* to state *Waiting for command* (recall Figure 5.8(a)) 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.

Figure 5.10 depicts our results. 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* 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

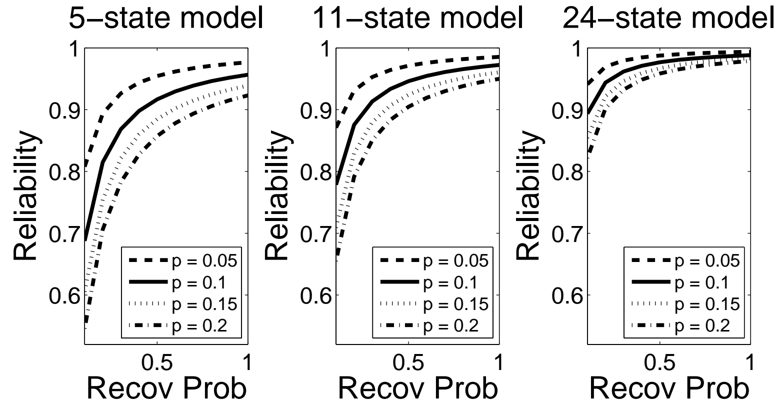


Figure 5.11: Analysis of sensitivity to models of different granularities of *DeSiController*

reliability ranges are larger when failure probabilities increase and/or recovery probabilities are lower. This observation agrees with what we observed in SCRover's *Controller*. 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.

Sensitivity to Model Granularity

Figure 5.11 shows the results of calculating the reliability of the *DeSiController* component based on its models at the three levels of granularity from Figure 5.8, with injected defect d_3 from Table 5.1 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.

The detailed model of *DeSiController* from Figure 5.8(a) is the one we have used in all of our measurements discussed in the preceding sections. Two higher-level models

of the same component, developed with the help of DeSi's designers, are depicted in Figures 5.8(b) and 5.8(c).

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 5.8(c)). Another observation is that the model from Figure 5.8(c) is more sensitive to recovery probability than the model from Figure 5.8(b), while the most detailed model (Figure 5.8(a)) is least sensitive. This agrees with the results obtained from SCRover's *Controller*.

In this more complex component, we observe 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 5.8(b)) 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*.

5.4 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 operational profiles. We have argued that these assumptions are not reasonable, and have presented a way to estimate operational profiles from different information sources.

We approached the challenges associated with the lack of information about a system and its components early in development by exploring the sources of information available at design time. Our evaluation and validation experiments indicate that, use

of our approach in determining operational profiles, results in accurate sensitivity in reliability estimates, where implementations are used as ground truth.

Chapter 6

Conclusions and Future Work

As our reliance on software systems grows, it has become more important to perform quality analysis early. This is because if problems are discovered after the system has been implemented, it is prohibitively expensive to mitigate the problems. In this dissertation, we have focused on addressing the following shortcomings of existing approaches in early software quality analysis: (1) the high cost of existing design-level reliability estimation approaches, especially when applied to modeling concurrent systems; (2) the high cost of testing-based approaches for performance analysis of third-party components, especially when testing at high workload; and (3) the unreasonable assumption on the availability of the system's and its components' operational profiles, which are typically gathered during runtime.

In this chapter, we summarize our contributions in Chapter 6.1, and highlight a few future work directions in Chapter 6.2.

6.1 Summary and Contributions

In Chapter 3 we proposed SHARP, an architecture-level, hierarchical framework that is capable of modeling *concurrent* systems in a *scalable* manner, without sacrificing the level of details we can model about the system. In SHARP, to generate a system model, first we generate models of the basic scenarios by leveraging system use-case scenario models. Then, we combine the models of the basic scenarios to form a higher-level model, according to the relationships between the lower-level models. Thus, system

reliability is the reliability of the highest-level scenario. This hierarchical approach is motivated by the fact that submodels are small, and that solving a number of smaller submodels is more computationally efficient than solving one huge model (as in "brute-force" approaches). Through extensive experimentation we validated the complexity and accuracy of this approach, which illustrates that the practical space and computational complexity benefits are achieved at the cost of small losses in accuracy as compared to existing techniques.

In Chapter 4, we presented a queueing model-based framework that accurately predicts response time of third-party WSs. Recall that performance testing is quite an expensive process, as it requires sending a large amount of requests to the system being tested, which may saturate its resource when testing at high workload. The main idea behind our approach is that we avoid testing at high workload, and instead use queueing models to guide extrapolation, so as to overcome the poor extrapolation results using standard regression analysis. We have shown that our approach is more accurate in extrapolation, while maintaining the accuracy of interpolation, as compared to applying standard regression analysis. Such information can be useful in WS selection (e.g., use the WS that provides the best performance), capacity planning (e.g., estimate how much traffic the system can handle), and traffic engineering (e.g., determine how much traffic should be sent to WSs that provide the same service).

In Chapter 5, we have overcome the lack of operational profile information by utilizing a variety of other available information sources. We have identified four major sources of information during design (Chapter 5.2): (1) expert's knowledge, (2) requirements document and system specifications, (3) functionally similar system/component,

and (4) simulation of architectural models, and apply the HMM-based technique proposed in [19] to estimate operational profiles from execution logs of a functionally similar system/component, or simulation logs. While our discussion has focused on component reliability analysis, we believe our approach is applicable to system-level reliability analysis as well. We have applied our operational profile estimation technique to the component reliability prediction process described in [19] to validate its effectiveness. We compared reliability estimates when the operational profile is estimated from different sources of information, and the results are validated by comparisons to an implementation-level technique, which is used as the "ground truth". 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.

6.2 Future Work

This section highlights a few directions to further improve software quality analysis.

6.2.1 Integrating Firmware Properties

In Chapter 5, while we have addressed the problem of estimating a component's operational profile in reliability analysis, the problem of not knowing the failure information remain unresolved. Estimating the failure information of a software component requires understanding of the underlying platform, such as the operating system, middleware, and hardware resources, which we collectively refer to as firmware. Existing software reliability analysis techniques, including [19] that we used in Chapter 5, assume the underlying firmware is reliable. Being able to determine the failure information with

more certainties allows us to focus on a smaller parameter space in studying a component's reliability (recall Chapter 5.2). At the same time, as discussed in Chapter 1, estimating a component's performance also requires knowledge about the firmware, as the component's performance is highly dependent on the performance of the firmware. Thus, in order to accurately predict a component's performance, it is important to model the firmware.

However, integrating firmware properties into software quality analysis is a challenging problem. This is because of the complex interactions between software and the underlying firmware. For example, how does one map an application-level operation (e.g., storing sensor data in a database) to a sequence of hardware-level operations (e.g., executing a sequence of CPU, memory and disk operations)? Such a mapping is typically needed in modeling the firmware, but this is a complex process as it involves going through multiple software and hardware layers.

Another challenging issue in integrating firmware properties is that firmware properties should be integrated in a *composable* manner, so that software designers need not to generate the model from scratch when they evaluate the same software on different firmware platforms. However, different firmware platforms may map the same application-level operation differently, and are therefore not composable. Thus, it is very expensive to study how the software system performs on different firmware platforms.

6.2.2 Performability Analysis

We treated performance and reliability separately in this dissertation; yet, the dependencies between software performance and reliability should not be ignored. This unified analysis is referred to as performability analysis in [50]. For example, a software system may have performance requirements (e.g., a request has to be completed in X seconds);

failure to meet such a requirement may be considered as a failure. Estimating how often this failure occurs involves estimating the system's performance. Another example is modeling systems that may run in a degraded mode: the system can provide its intended functionalities even when part of the system has failed, but the performance is degraded. For example, when a disk drive in a disk array has crashed, the software may still be able to read data from the disk array, but the response time may be higher.

One challenge is that existing performance estimation techniques focus on mean value-type analysis, while performability analysis typically requires knowing the distribution of performance metrics. For example, if a system is considered unreliable when it takes more than X to process a request, we need to know the distribution of the response time while integrating this requirement into a performability model. However, it is analytically difficult to obtain this information using existing performance analysis techniques (such as mean value analysis in QNs [69]). One can use simulation to gather this information, but simulation is more expensive.

Another challenge is that a system's performance is highly dependent on the firmware it runs on, and thus we face a similar challenge as discussed above. Without accurate performance estimates, performability analysis would not be very meaningful.

6.2.3 Reliability Testing

As discussed in Chapter 1, there are few alternatives to reliability testing to evaluate software reliability where the source code is unavailable. As in performance analysis, analyzing the reliability of third-party components is important, because they contribute to the reliability of the overall system. Hence, it is vital to understand the reliability of third-party components before it has been integrated into the system.

The challenge is that reliability testing is prohibitively expensive, because, on the average, it requires sending a large amount of requests (in the order of 100,000 requests)

to observe one error. The Seekda WS search engine [5] estimates the reliability of a WS by monitoring the responsiveness of the WS server (e.g., if it responds to `ping`). While this can be used as a coarse estimate, the WS itself has never been evaluated. For example, Seekda would report a WS as reliable even if the WS returns incorrect results. The work in [80] proposes a way to estimate reliability of third-party WSs using failure data of "similar" service users (e.g., users from the same ISP or making requests to WSs in the same administrative region). We argue that this approach may be inaccurate because different users may have different operational profiles and/or reliability definitions (recall Chapters 3 and 5).

Bibliography

[1] http://webgis.usc.edu/Services/Geocode/WebService/GeocoderService _V02_94.asmx?WSDL.

[2] http://vista.usc.edu/ lccheung/reliability.

[3] Java adventure builder reference application. http://adventurebuilder.dev.java.net.

[4] SCRover. http://cse.usc.edu/iscr/pages/ProjectDescription/home.htm.

[5] Seekda. http://webservices.seekda.com.

[6] TPC-App benchmark. http://www.tpc.org/tpc_app/default.asp.

[7] Weather bug web service. http://api.wxbug.net/webservice-v1.asmx?WSDL.

[8] Web service description language (WSDL). www.w3.org/TR/wsdl.

[9] CSCI 477. Design and construction of large software systems, University of Southern California, 2003. http://sunset.usc.edu/ neno/cs477_2003/.

[10] S. Balsamo et al. Model-based performance prediction in software development: A survey. *IEEE TSE*, 30(5), May 2004.

[11] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, 1975.

[12] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance prediction of component-based systems — a survey from an engineering perspective. *Architectung Systems with Trustworthy Components*, 2006.

[13] Christopher Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[14] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34:135–137, January 2001.

[15] Barry Boehm, Jesal Bhuta, David Garlan, Eric Gradman, LiGuo Huang, Alexander Lam, Ray Madachy, Nenad Medvidovic, Kenneth Meyer, Steven Meyers, Gustavo Perez, Kirk Reinholtz, Roshanak Roshandel, and Nicolas Rouquette. Using empirical testbeds to accelerate technology maturity and transition: The scrover experience. In *Proceedings of ISESE'04*, pages 117 – 126, 2004.

[16] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and Francesco Lo Presti. Flow-based service selection for web service composition. In *IEEE ICWS*, 2007.

[17] Senthilanand Chandrasekaran, John Miller, Gregory Silver, Budak Arpinar, and Amit Sheth. Performance analysis and simulation of composite web services. *Electronic Markets*, 13(2), 2003.

[18] Leslie Cheung, Leana Golubchik, and Nenad Medvidovic. SHARP: A scalable approach to architecture-level reliability prediction of concurrent systems. In *Proceedings of the First International workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, May 2010.

[19] Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *ICSE 2008*.

[20] R.C. Cheung. A user-oriented software reliability model. *IEEE TSE*, 6(2), 1980.

[21] Vittorio Cortellessa and Vincenzo Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *CBSE 2007*, pages 140–156, 2007.

[22] Vittorio Cortellessa and Raffaela Mirandola. Deriving a queueing network based performance model from UML diagrams. In *ACM Proc. Intl Workshop Software and Performance*, pages 58–70, 2000.

[23] Vittorio Cortellessa, Harshinder Singh, and Bojan Cukic. Early reliability assessment of UML based software models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 302–309, 2002.

[24] Andrea D'Ambrogio and Paolo Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In *WOSP'07*, 2007.

[25] Carl de Boor. *A Practical Guide to Splines*. Springer, 2001.

[26] Norman Draper and Harry Smith. *Applied Regression Analysis*. Wiley-Interscience, 1998.

[27] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35:202–211, November 2000.

[28] Rehab El-Kharboutly, Reda A. Ammar, and Swapna S. Gokhale. UML-based methodology for reliability analysis of concurrent software applications. *I. J. Comput. Appl.*, 14(4):250–259, 2007.

[29] S. Gokhale and K. Trivedi. Analytical models for architecture-based software reliability prediction: A unification framework. *IEEE Transactions on Reliability*, 55(4), Dec 2006.

[30] Swapna Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE TDSC*, 4(1), Jan 2007.

[31] Swapna Gokhale, Michael Lyu, and Kishor Trivedi. Reliability simulation of component based software systems. In *Proceedings of ISSRE'98*, pages 192–201, 1998.

[32] Swapna Gokhale and Kishor Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *ISSRE 2002*.

[33] Swapna S. Gokhale, W. Eric Wong, Kishor S. Trivedi, and J. R. Horgan. An analytical approach to architecture-based software reliability prediction. In *IEEE International Computer Performance and Dependability Symposium*, 1998.

[34] K. Goseva-Popstojanova and K. Trivedi. Architecture-based approaches to software reliability prediction. *Intl. J.Comp. & Math. with Applications*, 46(7), Oct 2003.

[35] Katerina Goseva-Popstojanova, Ahmed Hassan, Walid Abdelmoez, Diaa Eldin M. Nassar, Hany Ammar, and Ali Mili. Architectural-level risk analysis using UML. *IEEE TSE*, 29(3), Oct 2003.

[36] Katerina Goseva-Popstojanova and Sunil Kamavaram. Software reliability estimation under uncertainty: Generalization of the method of moments. In *Proc. of HASE 2004*.

[37] Katerina Goseva-Popstojanova and Kishor Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45:179–204, 2001.

[38] Anne Immonen and Eila Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, Jan 2007.

[39] S. Krishnamurthy and A. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *In Proceedings of ISSRE 1997*.

[40] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 48–63, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] B. Littlewood. A reliability model for Markov structured software. In *Proceedings of the international conference on Reliable software*, pages 204–207, 1975.

[42] B. Littlewood. Software reliability model for modular program structure. *IEEE Transactions on Reliability*, 28(3), 1979.

[43] J. Magee and J. Kramer. *Concurrency: State Models And Java Programs*. John Wiley & Sons, 2006.

[44] Sam Malek, Nels Beckman, Marija Mikic-Rakic, and Nenad Medvidovic. A framework for ensuring and improving dependability in highly distributed systems. *Architecting Dependable Systems III*, Oct 2005.

[45] Sam Malek, Chiyoung Seo, Sharmila Ravula, Brad Petrus, , and Nenad Medvidovic. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *ICSE 2007*.

[46] Moreno Marzolla and Raffaela Mirandola. Performance prediction of web service workflows. In *QoSA'07*, 2007.

[47] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering*, 26(1):70–93, Jan 2000.

[48] Daniel Menasce, Honglei Ruan, and Hassan Gomaa. QoS management in service-oriented architectures. *Performance Evaluation*, 64(7-8), 2006.

[49] C. D. Meyer. Stochastic complementation, uncoupling markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2)::240–271, 1989.

[50] J.F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, 29:720–731, 1980.

[51] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[52] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.

[53] D.E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40–52, 1992.

[54] Erik Putrycz, Murray Woodside, and Xiuping Wu. Performance techniques for cots systems. *IEEE Softw.*, 22(4):36–44, 2005.

[55] L.R. Rabiner. A tutorial on hidden markov models. *Proceedings of the IEEE*, 77:257–286, 1989.

[56] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service slas. In *Proceedings of FSE-16*, 2008.

[57] Carl Edward Rasmussen and Chris Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[58] R.R. Reussner, H.W. Schmidt, and I.H. Poernomo. Reliability prediction for component-based software architectures. *J. of Sys. and Software*, 66(3), 2003.

[59] Genaina Rodrigues, David S. Rosenblum, and Sebastin Uchitel. Using scenarios to predict the reliability of concurreny component-based software systems. In *FASE 2005*.

[60] Roshanak Roshandel, Somo Banerjee, Leslie Cheung, Nenad Medvidovic, and Leana Golubchik. Estimating software component reliability by leveraging architectural models. In *Emerging Results track, ICSE 2006*, pages 853–856, May 2006.

[61] Roshanak Roshandel and Nenad Medvidovic. Multi-view software component modeling for dependability. *Architecting Dependable Systems II*, 2004.

[62] Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. A Bayesian model for predicting reliability of software systems at the architectural level. In *QoSA 2007*.

[63] Roshanak Roshandel, Bradley Schmerl, Nenad Medvidovic, David Garlan, and Dehua Zhang. Understanding tradeoffs among different architectural modeling approaches. In *WICSA 2004*.

[64] P. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *International Conference on Stochastic Control and Optimization*, Amsterdam, 1979.

[65] M Shooman. Structural models for software reliability prediction. In *In Proceedings of ICSE 1976*.

[66] Kyle Siegrist. Reliability of systems with Markov transfer of control. *IEEE TSE*, 13(7), July 1988.

[67] Connie Smith. *Performance Engineering of Software Systems*. Addison Wesley, 1990.

[68] Hyung Gi Song and Kangsun Lee. sPAC (web services performance analysis centre): Performance analysis and estimation tool of web services. *BPM 2005, LNCS 3649*, 2005.

[69] W. Stewart. *Probability, Markov Chains, Queues, and Simulation*. Princeton University Press, 2009.

[70] William Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[71] Richard Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[72] H.C. Tijms. *Stochastic Models*. John Wiley and Sons, 1994.

[73] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. *SIGSOFT Softw. Eng. Notes*, 26(5):74–82, 2001.

[74] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33(1):291–302, 2005.

[75] Kaiyu Wang and Naishio Tian. Performance modeling of composite web services. In *Proceedings of the Pacific-Asia Conference on Circuits, Communications and System*, 2009.

[76] W. Wang, D. Pan, and M. Chen. Architecture-based software reliability modeling. *J. of Systems and Software*, 79(1), 2006.

[77] M. Xie and C. Wohlin. An additive reliabiltiy model for the analysis of modular software failure data. In *ISSRE 95*.

[78] Sherif M. Yacoub, Bojan Cukic, and Hany H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, 2004.

[79] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2):292–333, 1997.

[80] Zibin Zheng and Michael R. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 35–44, New York, NY, USA, 2010. ACM.