PERFORMANCE ENGINEERING OF SOFTWARE WEB SERVICES AND
DISTRIBUTED SOFTWARE SYSTEMS

Chia-en Lin

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

May 2014

APPROVED:

Krishna Kavi, Major Professor
Robert Brazil, Committee Member
Kathleen Swigger, Committee Member
Robert Akl, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Mark Wardell, Dean of the Toulouse Graduate School
To meet the requirements and facilitate using Web services, in this work a WSDL extension to permit specification of non-functional or quality of service (QoS) properties is first proposed. Additionally, a QoS-aware framework is established to adapt publicly available tools for Web services, augmented by ontology management tools, along with tools for performance modeling to exemplify how the non-functional properties such as response time, throughput, or utilization of services can be addressed in the service acquisition and composition process. The framework is extended with additional qualitative information to the service descriptions using Business Process Execution Language (BPEL), which can be used to explore design options, and have the QoS properties analyzed for the composite service.

The main issue in my research is performance evaluation in software system and engineering. Research into Web service computation is the first half of this dissertation, while performance antipattern detection and elimination is the second part. In this work, I analyze performance antipatterns to extract detectable features, influential factors, and resource involvements so that we can lay the foundation to detect their presence. I propose a system abstract layering model and suggest profiling methods for performance antipattern detection and elimination. Solutions proposed can be used during the refactoring phase, and can be included in the software development life cycle. Proposed tools and utilities are implemented and their use is demonstrated with RUBiS benchmark.
Copyright 2014

by

Chia-en Lin
ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my major advisor Dr. Krishna Kavi for his continuous support of endless patience and knowledge in guiding me throughout research projects and thesis. With his experienced insight, I have had the opportunity to approach research problems from different aspects, and to explore potential solutions in my own way. On this long road of research, one cannot wish for a greater and more consistent supervisor and mentor in guidance.

I also owe thanks to my committee members: Dr. Robert Brazile, Dr. Kathy Swigger, and Dr. Robert Akl. They all devoted their time to help shape my dissertation, and generously pointed my research in the right direction. Their encouragement is one of the keys that made the thesis possible.

My sincere thanks goes to Dr. Shih-Kun Huang, of National Chiao-Tung Univeristy in Taiwan. He generously accepted my request for guidance in the latter years of my dissertation. He faithfully met online with me weekly over the years, and kindly gave excellent advices in all ways possible. I was very lucky to have him as an academic partner and friend who lit the road for me.

I am grateful for my colleagues and coworkers in the lab and the department. Without their friendship and support over the years, the daily routine of research would be harder.

My family is the cornerstone that has supported me throughout the years. I thank my mom for having me in her prayers every single day, and believing in what I was pursuing; my sister Natalie who inspired and supported me fully with great divine love along with my brother-in-law Frank; my sister Jane who is always there for me no matter what I have been through; my niece Joy gives the best assistance ever when I need her help. Words are not enough to express my thankfulness to them. Finally and most importantly, I thank my wife Joy for her love to me and faith in God throughout the years in every moment together. This would be impossible without her company and her taking care of Ethan and me.
CHAPTER 4. QOS-AWARE BPEL WEB SERVICE SELECTION AND COMPOSITION

4.1. Introduction 28
4.2. QoS-Aware Framework in SOA 29
  4.2.1. Framework Description 29
  4.2.2. QoS-Aware Framework with WS-BPEL Extension 31
4.3. Service Compositions with WS-BPEL in the QoS-Aware Framework 33
  4.3.1. Business Process Service Orchestration 34
  4.3.2. Business Process Service Choreography 35
4.4. From WS-BPEL to LQN 37
  4.4.1. WS-BPEL Constructs 38
  4.4.2. Layered Queueing Networks 38
  4.4.3. Transition Rules from BPEL to LQN 39
  4.4.4. Data Dependency in Transformation 40
4.5. Case Studies 42
  4.5.1. Facial Detection and Recognition Example 43
  4.5.2. Data Dependency Considerations 47
  4.5.3. Performance Space Considerations 51
4.6. Conclusion 52

CHAPTER 5. PERFORMANCE ANTIPATTERN DETECTION AND SOLUTION SUGGESTIONS

5.1. Performance Antipattern as Design Patterns 53
5.2. Design Processes with Performance Antipattern 61
5.3. Performance Baseline 67
5.4. Suggestive Profiling 73
  5.4.1. Path Profiling 75
  5.4.2. Thread Behavior Profiling 75
  5.4.3. Network Profiling 76
## LIST OF TABLES

3.1 Web Service Interfaces of WSOE ........................................... 25
3.2 Layered Queueing Model for Facial Recognition Service Composition Example .................. 27
3.3 Layered Queueing Model for Facial Recognition Service Composition Example .................. 27
4.1 Mapping BPEL Basic Activity to LQN Elements ........................................... 41
4.2 Mapping BPEL Structured Activity to LQN Elements ........................................... 42
5.1 Performance AntiPatterns with Detectable Features ........................................... 59
5.2 Performance AntiPatterns with Undetectable Features ........................................... 60
5.3 System Abstraction Layers ........................................................................ 70
5.4 Suggestive Profiling Method in the System Abstract Layer Context ......................... 74
6.1 RUBiS Request Types with Identification Numbers ........................................... 86
# LIST OF FIGURES

3.1 QoS-Aware WSDL Schema for Performance Parameters 17  
3.2 Performance Ontology Model 20  
3.3 QoS Ontology Model with Refined Ranks 21  
3.4 Web Services Layered Queueing Network Modeling 24  
4.1 QoS-Aware Service Oriented Architecture Modeling 30  
4.2 Facial Detection BPEL (FD) 44  
4.3 Image Converter BPEL (IC) 44  
4.4 Facial Recognition BPEL (FR) 44  
4.5 Facial Detection LQN Model 45  
4.6 Image Converter LQN Model 46  
4.7 Facial Recognition LQN Model 46  
4.8 FD IC FR Sequential Composition LQN Model 47  
4.9 Atomic Composition of FD, IC, and FR 47  
4.10 Choreography Topology for Process Service Composition of FD, IC, and FR 48  
4.11 FD IC FR Choreography Model 49  
4.12 IC Service Execution Time vs If-Clause Probability 50  
4.13 IC Utilization vs If-Clause Probability 50  
4.14 FR Service Execution Time vs Database Probability 50  
4.15 FR Utilization vs Database Probability 50  
4.16 IC Service Execution Time vs If-Clause Probability on Service Design Alternatives 51  
4.17 IC Utilization vs If-Clause Probability on Service Design Alternatives 51
5.1 Performance Modeling Life Cycle with Performance Antipattern Refactoring 63
5.2 Refined Performance Modeling Life Cycle with Performance Antipattern Refactoring 65
5.3 Overview of Performance Baseline and its Relation to Performance AntiPattern 69
6.1 RUBiS Benchmark Tiers 84
6.2 RUBiS Benchmark Using browse_only_15 Workload 87
6.3 RUBiS Benchmark Using default_transition_15 Workload 88
6.4 One Web Server Dispatches Requests to Two Application Servers in RUBiS Noted with Performance Metrics 89
6.5 Dispatched Number of Requests to worker1 and worker2 of JBoss 92
6.6 Average Execution Time of Requests Dispated to worker1 and worker2 of JBoss 92
6.7 RUBiS Benchmark with 800 Users Using Various Sets of MPM Configuration 94
6.8 Disassemble Snapshop of ap_process_request_internal() Function in httpd 96
6.9 Snapshot of Request Flow Graph of ap_process_request_internal() Function in the httpd Web Server 97
CHAPTER 1

INTRODUCTION

1.1. Motivations

Performance analysis in software system is complex because of the number of components and the interactions among them. A computing environment may be composed of general purpose servers such as proxies, databases, and the web. The servers may be customized to host applications in wireless, telecommunication, or supercomputing applications. The servers run on a platform with either an open or proprietary operating system that is supported by either virtual or physical architecture. Under the system architecture, hardware components with different capacities and organizations contribute to the already large dynamics of the system. In addition, software components running on a system are already very complex. Each of these aspects of a system requires sophisticated attention both in developing and deploying phases. On some occasions, components that work well in isolation may have issues when they are paired to work together. Developing and deploying a full-fledged system with the expected performance behavior may require multiple domains of knowledge about both the system and its applications.

Both the system and the software provide services to other applications through communication mechanisms. A service is a unit of functionality which can be accessed by internal peers or external counterparts through internet. As inter-networking becomes the default platform of modern computing, Web services have become the new form of software components relying on service orientated architecture paradigm. The new paradigm adds to the complexity of system engineering. We have to wonder if the performance of services can be evaluated using the same approaches used for component-based software. Despite the existing hurdles in evaluating the integration of software components, service composition poses new paradigms in its own right. Service components face a more dynamic environment which demands more flexibility in cooperation. It turns out that not only do we have to know the limitations of a service, but we also need to see the potential of
the service within other services. The dynamics of the services provide different levels of services in selections of quantity and quality. In short, service is different from a traditional software architectures in that it’s autonomous, self-described, reusable, and highly portable. In addition to conventional performance evaluation technique, approaches and tools have to support these additional features.

The works in this thesis are to tackle performance evaluation in which both fields mentioned above. Performance evaluation approaches and tools used to solve the problem in each individual aspect have been studied. The goal of this work is to recognize the system architecture and the capability of components in practice, and provide solutions to performance evaluation with methodologies and tools. The purpose of performance evaluation is to diagnose problems, and then fix them. Trouble shooting a problem used to be a highly manual task. It also demands expertise and familiarity in application domains to be able to address the root causes. When the cause has been spotted, there are still more questions about how to solve it. When a solution is not available, engineers have to come up with a diagnosis that can remedy the anomaly. When a solution exists, engineers have to match it to the problem and make it work. They also have to worry about the effectiveness of the applied solution. The process is extremely tedious and time consuming.

In the service-oriented computing field, the number of services increases only monotonically as more systems provide services online. The purpose of preliminary performance troubleshooting is to determine whether or not the service lives up to the agreed capacities and capabilities. The solution discovery involves performance estimation of services. The large number of potential services available for consideration only makes the activity more complex. The goal of this research is to provide a framework to ease the complexity of performance evaluation, and to automate the process with the help of appropriate tools.

In the following sections, related problems that engineers may face in terms of performance evaluation in software systems are illustrated. The approaches documented here proposes possible solutions and their respective design philosophies. Contribution summaries and the roadmap of this thesis are presented.
1.2. Approaches

To both software systems in general and to the Web service computations, our research proposes a system layer model to describe system components, their functionality, and behaviors in general. We adapt a drill-down analysis method that involves digging into layers of software and hardware of the system stack. The setting enables us to pinpoint elements in the system more precisely and flexibly. Instead of identifying a software entity or component in the system with the name and its functionality, we can also address the refined context in which it is situated. This approach enables us to freely moving from high-level view to deeper details. Mappings between component entity and resource allocations can lead to clues of the causality especially when determining the root cause of a performance bottleneck. We also emphasize the feasibility and practicality of our approach in addressing real problems. The context information can be shown through mapping, in which the precise quantity of both logical and physical resources are accounted for.

Our approach also stresses the compatibility of the existing processes. In software engineering, software life cycle and system development are guided by regulated processes which, if followed, assert the qualified system results. In software performance engineering, modeling activities are guided in order for performance testing and measurement to be successful. To facilitate these processes with our frameworks, our approaches consider their suitability when building our tools. Essentially, our approach opens opportunities to adapt other frameworks when following the standards in the field. We expect our framework to be backward-compatible so that users who do not adapt our approaches can still use our frameworks and tools.

Analysis phase in both system and its performance modelings is an essential step in every software development. It enables the understanding of system behavior with mathematical models and predictive performance outcomes by solving them. During the modeling, appropriate assumptions are needed to produce results. The precision of the assumptions can impact the precision of the prediction. If more realistic information is available, the reliability of the prediction will improve. To do so, data has to be gathered before the prob-
lem is identified and analyzed. Our approach is capable of handling data from real system monitoring or simulation testing. In the process of performance debugging, data can be compared and analyzed to seek better solutions.

Performance solutions should be documented and stored as assets to be reused to save the debugging cost. In software engineering, users follow design pattern templates to solve the issues using best practices. In service oriented architecture, service information is archived so that users can search for candidates given the functional description. Newly composed services can also be recorded for future retrieval. Our approach in performance evaluation also promotes reuse in which best practice knowledge can be applied in the engineering process.

1.3. Contributions and Roadmap

The thesis can be viewed in two parts: one in Web service computing, and the other in performance antipattern modeling. Before going into the details of the works, we highlight the key points of both fields of the research:

In performance evaluation in Web services compositions:

- WSDL extensions for specifying nonfunctional properties
- Ontology modeling for classification of non-functional properties
- Framework for the discovery of services that meet both functional and non-functional requirements
- Framework for computing performance (stochastic) measures of composed services
- Framework extension to BPEL that permits the description of QoS properties with services, reasonable composition processes of these services, and the generation of models for deriving QoS properties of composed services

In performance antipattern detection and elimination:

- Identifying detectable features, influential factors, and involved resources in software performance antipattern
Propose performance baseline and system abstract model for performance debugging in software systems

Propose suggestive profiling approaches and tools to detect performance antipattern and generate solution feedback of performance modeling

Building framework tools and utility for real production systems

The rest of the dissertation consists of the following chapters: Chapter 2 presents the background and the related works. Chapter 3 describes the inefficacy of handling non-functional properties of services using WSDL. We approached the problem by adding quality awareness with WSDL, and introduced ontological and performance modeling to work with non-functional properties of Web services and their composition. Chapter 4 describes a full framework, extended from the previous QoS-aware framework by adding the capabilities of using BPEL to model business logic. Methodologies of standard Web service compositions, such as orchestration and choreography, are adapted and reasoned to be compatible with our framework. Within the framework, techniques to translate service representations from BPEL to layered queueing models are presented. Chapter 5 tackles the performance evaluation from different aspects using software performance antipatterns. Performance baseline is introduced for preliminary performance debugging. System layered structure and suggestive profiling method laid the foundation for performance antipattern detection and solution feedback. Suggestive profiling tool and its framework utilities are implemented. Processes of using the tool in the performance modeling refactoring phase are presented. Chapter 6 describes the RUBiS benchmark used in our testing. Case study of performance antipattern detection and solution suggestion methods using unbalanced processing, more is less, and god class antipatterns are presented. Finally, Chapter 7 presents the summary of the dissertation.
CHAPTER 2

PERFORMANCE EVALUATION RELATED WORKS

In this chapter, research that is closely related to this work on Web service composition and performance anti-pattern detection is discussed. The first section surveys approaches to quality-of-service (QoS) annotations, Web service compositions using service-oriented architecture and Business Process Execution Language (BPEL). The second section surveys methods for detection, diagnosis and elimination of performance anti-patterns.

2.1. Web Services and BPEL Composition Performance Evaluation

Web-service composition allows users to build more complex services from available services. The current techniques assumes that user is knowledgeable of issues such as service portfolios and ontologies, in order to compose services. However, often users are not given information about non-functional properties such as performance, reliability and security. And tools for composing services based on non-functional properties are lacking. The next section surveys the currently available methods for service composition and QoS extensions to service description languages such as WSDL.

2.1.1. SOA and Web Services Composition

The description of non-functional properties related to service oriented architecture (SOA) operational management has been described in [31]. In addition to adding some QoS criteria, semantic interpretation of the extensions have been discussed in various frameworks [20] [60] [48]. An approach to describing service lifecycle information and QoS guarantees offered by a service based on OWL-S can be found in [70]. Here, service profiles are appended with QoS characteristics to generate a corresponding service description repository. The OWL-S based repository can automatically cover the traditional UDDI registry by mapping its elements. In [64], WSDL is extended to X-WSDL where non-functional criteria are added in service definition. Following its predecessor X-UDDI [63], the Web service registration and publication can be queried on the basis of this criteria. In [68], a unified semantic Web
services publication and discovery framework are proposed with a QoSMetrics extension to WSDL using PS-WSDL, USQL for service query, and UDDI mapping suites. In this work, the focus is on a proof of concept for WSDL extension and its correspondent non-functional semantic model engineering, but not on the service registration. With our framework, it should be straightforward to apply well-defined UDDI extension tools such as mentioned in [5], or other registry tools.

To enable semantic description of service extensions, several ontological languages have been proposed. An overview of some of these languages can be found in [66]. They focus on the semantic modeling and mapping ontology applied to service descriptions. Our framework focuses on the engineering of the ontology model and its references to the performance modeling tools. With the help of ontology mapping, different service descriptions and advertisement standards should be easy to adapt in our framework.

Service composition methods and their languages can be broadly categorized into different types: orchestration, choreography, coordination, and assembly [12]. While emphasizing different aspects to approach the issue, composition methods use ontology to annotate QoS attributes that provide common ground for service synthesis, execution, and adaptation [30]. In QoS-aware service composition, services are selected based on inter and intra task constraints. They can also be grouped into deterministic and non-deterministic tasks depending on when these attributes were made known [12]. Various research approaches are hoping to gain optimal results by using detailed descriptions of QoS values of services during composition [18] [83]. In [86], a quality-driven middleware serves as a composition manager that models multidimensional QoS attributes with utility functions, and optimizes them by local selection and global planning for different quality criteria.

In [58], requested and provided QoS properties are respectively expressed as specification documents and service specification documents in the open dynamic execution environment. The framework serves as a broker for service compositions that utilizes the QoS model in its own ontological language. Service selection algorithms and metrics based on the ontology are utilized by the service broker. Its objective is to support ad-hoc service collabor-
orations, while our approach is to facilitate the description of QoS properties of existing and new composite services. The work is similar to ours with the emphasis on using ontology model as the tool to reason QoS attributes semantically. When monitoring the execution condition, the ontology model can facilitate the selection of the correct set of QoS values according to the execution environment. The QoS-aware ontology modeling framework proposed in the thesis can serve the same purpose.

The advantage of proposed framework in this research is in its facilitation of stochastic performance evaluation during service composition. The above mentioned related works do not consider the use of ontology and performance models when addressing the evaluation of QoS properties of service composition. The framework also considers the use of different performance tools with the model-related elements in the ontology, facilitating the usage of QoS attributes based on context and selecting appropriate models and tools for ascertaining properties during composition.

2.1.2. BPEL Composition and Performance Evaluation

The promise of service oriented computing, and the availability of Web services in particular, promote delivery of services and creation of new services from existing services [62] – service components are assembled to achieve integrated computational goals. Business organizations strive to utilize the services and provide new service solutions; to achieve these goals, they will need appropriate tools [13]. As webs and internet based services grow into Clouds, inter-dependency of services and their complexity increases tremendously. The Cloud ontology suggested in [85] depicts service layers from a high-level, such as application and software, to a low-level, such as infrastructure and platform. Each component resides at one layer can be useful to others as a service. It hints the amount of complexity resulting from not only horizontal but also vertical integrations in building and deploying a composite service. Our framework tackles the complexity of the selection and composition issues with additional qualitative information to the service descriptions in BPEL. Engineers can use BPEL to explore design options, and have the QoS properties analyzed for the design. QoS properties of each service are annotated with our WSDL extension for future references.
There have been several works on QoS-awareness for BPEL services. In [19], a service broker offers composite service with multiple QoS classes to the users. The selection scheme optimizes aggregated QoS of incoming request flows using linear programming. In [23], business workflow is parsed into a tree structure, where heuristic algorithms are applied to select service candidates based on QoS properties. In [57], QoS is acquired by constructing an activity graph and reasoning with the dependencies among them for the QoS parameters, including response time and cost. A declarative approach is proposed in [9] by creating the policy-based language QoSL4BP to specify QoS constraints and logic over scopes of the orchestration. QoS planning, monitoring, and adaptation of the BPEL can be expressed to model the service behavior. An extension to BPEL for specifying QoS and non-functional requirements is proposed in [3]. The extension point is located at the service invocation of a partner Web service. Our framework is able to provide compatible SOA infrastructure to test different approaches surveyed above and others, however, the foundation to address QoS properties for BPEL relies on the WSDL extension at the service level [50]. The benefit is that modeling business services to annotate QoS properties is compatible with standard WS-BPEL without the need to introduce other artifacts.

Performance evaluation with BPEL often involves analytical models by transforming the business logic into appropriate model logic. In [77] and [17], BPEL processes are translated into stochastic Petri nets by a set of rules that model waiting queues and their performance distributions. In [11], a formalism for the SYNTHESys framework [39] is generated by the translation from BPEL to PerfBPEL models. The PerfBPEL serves as the performance annotation to the BPEL workflow, and a Markov chain for the model can be generated. Then multi-formalism modeling technique enables the use of other tools for analysis. In [54], BPEL is annotated with a performance metadata for operations and resources. A queueing model can be derived from these annotations to compute throughput and response times. While the translation is similar to ours, our framework uses an ontology for QoS data management, and uses LQN to keep the original mapping of service architecture. In [7], support from abstract to executable processes for service orchestration is proposed.
according to three levels: needed functionality, expected QoS, and composition flow. Process realization, discovery, classification, and selection steps lead to the composition. The expected QoS is determined by a classification method to select services for composition. While our framework can also rank services using ontology models and plug in different selection filters, the QoS prediction for service composition is based on the result of modeling analysis.

A feature-completed Petri net semantic counterpart for BPEL has been established in \[52\]. As mapping from BPEL is easily obtained, Petri net can be subjected to formal model checking \[80\] and workflow performance analysis \[81\]. Our work allows for performance evaluation of composed services by transforming BPEL to LQN, such that the LQN models can be analyzed to obtain relevant performance properties.

2.2. Performance Anomaly Detection and Root Cause Analysis

Design pattern is a reusable solution to a common recurring problem. The solution to the problem described in the pattern serves as a best practice template for a practitioner to use. An antipattern is similar to a pattern, identifies a problem and a possible solution. The solution suggests what to avoid under certain circumstances in order for the design to be a best design. Spotted antipatterns in the system can be eliminated to improve quality. In the same fashion, performance antipattern helps practitioners in checking performance pitfalls hidden, preventing the system from obtaining best possible performance. In the following subsections, approaches related to performance antipattern detection, diagnosis, and solutions that are closely related to our work are discussed and compared with our methods.

2.2.1. Performance Antipattern Detection

The concept of antipattern has been proposed more than a decade ago \[16\]. The main source of technology independent software performance antipatterns have been published in \[72\] \[73\] \[75\]. Performance antipattern detection has been addressed in different systems and models. Performance detection in component based enterprise systems has been proposed
in [65], where a rule-based performance diagnosis tool is presented. The tool can work with EJB applications, in which data from runtime systems is extracted and applied with rules for antipattern detection. The method is limited to EJB systems, which can not be applied to general servers. Another performance detection and solution approach presented in [78] discusses the performance antipattern in the context of the Palladio Component Model (PCM) [15] software architecture modeling language. A queueing model is derived from the software model in PCM, and is solved to generate performance indicators. The predictive values are matched against performance antipattern rules in PCM to determine whether antipattern exists. Once detected, analyzed solutions in PCM can be applied. It also proposed iterative processes to solve antipattern one by one. A similar approach but using Architecture Description Language (ADL) based architecture can be found in [24]. Approaches using software architecture modeling in performance evaluation is promising. However, these approaches are limited to the model-based approaches with a specific architecture language. In [25], performance antipatterns are presented using logical predicates. The problem description for an antipattern is interpreted and presented using first order logic equations. The approach focused on antipattern presentation and detection for generation of feedback. The approaches from the above do not address how to interpret threshold values in the problem and solution representations of performance antipattern. In this work, approaches that can be easily applied in real practice will be emphasized. The framework of this work also provides mechanism to collect and analyze threshold values applicable to run-time systems.

2.2.2. Automatic Diagnosis and Feedback Generation

In [84], a rule-based automatic software performance diagnosis framework is proposed for bottleneck detection. Layered queueing models are used to construct performance model for performance prediction. The solved performance indices are checked against predefined rules to detect performance bottlenecks. The rules will also suggest mitigation rules to reduce operations and add resources, once bottlenecks are detected. The solution feedback is largely dependent on the definition of the rules. The success of the system depends on the extensibility of the rules. The feedback solution depends on translating performance
model attributes into design context, which has no specific rules for practitioners to follow. A similar approach is presented in [79], which extracts software and system architecture and creates a queueing model for performance anomaly detection. In the feedback process, the architecture model is used for redesign considerations. An approach is proposed to address these concerns about detection and automatic feedback with real system thresholds so that performance antipatterns can be applied in real practice. In [26], a special detection approach in finding the most guilty performance antipattern is proposed. The process checks performance antipattern symptoms against system requirements, and filters out the ones that do not violate them. The final list of performance antipatterns are ranked using scores calculated from equations defined for specific performance criteria. In this work, utilities in ranking the performance antipatterns are not provided. With the feature of the framework, it is left for practitioners to determine which specific symptoms are most malicious to be focused on first.

2.2.3. Solution Suggestions

A solution suggestion takes many forms and approaches in the literature. Our discussion focuses on the ones that are related to performance anomaly detection and root cause analysis. In [41], the performance anomaly clustering method is used to narrow down suspicious components in the distributed systems. Clusters are used to chain components together when they are affected by the same root causes. The clustering establishment is based on the similarity of the performance measurement. To identify the performance problematic spots, relationships between groups of clusterings are compared. The findings of performance anomaly are at the level of either a server or a method level. Further diagnosis steps will need to rely on the practitioner’s system knowledge. In [76], a framework for depicting performance anomaly manifestations is proposed with controlled configuration settings. By studying performance anomaly manifestations by using controlled configurations as root causes, design space can be classified with its unique configuration and whether or not performance problems exist. The coverage of the approach depends on the number of controlled configuration used. In practice, it is not feasible that every configuration and
manifestation can be covered. Our approach collects the data from the software and system, and establish the performance measurement specifically reflecting the real scenarios of the system under performance debugging. To discover the root causes, systematic processes are proposed which provide suggestive performance anomaly solutions.
CHAPTER 3

WEB SERVICE COMPOSITION USING QOS PROPERTIES

3.1. Introduction

Service oriented architecture (SOA) offers a flexible methodology for the creation and management of software services. Software services are well-defined business functionalities situated in loosely-coupled and distributed computing settings such as Cloud and Web. Each service provides a specific and well defined functionality. Well defined interfaces permit for the discovery and invocation of services. Web service is a realization of the SOA concept. Available standards allow for the creation, registration, discovery and invocation of Web services. Web Services Description Language (WSDL) can be used to specify the functionality of a service along with its communication protocols. Service providers can register services with Universal Description Directory and Integration (UDDI) or other such registry services. Service repository can be queried by customers to discover needed services. The discovery of a service is based on searching through categories and by matching the specification given in WSDL.

The goal of our research is to discover services based not only on their functionality but also based on non-functional (or quality of service) properties. In addition, our goal includes service composition and specification of non-functional properties of composed services. These goals require the ability to specify non-functional (or QoS) properties with services, and the ability to compute non-functional measures of composed services. Ascertaining certain non-functional properties of composed service require models and tools that are appropriate for the specific property (e.g., stochastic models for performance measures). In this chapter, we explore the development of the necessary framework for composing performance properties using queuing models.

WSDL can only be used for specifying functionality of services. Non-functional properties, including several quality of service (QoS) characteristics, are crucial to the success and wider adoption of Web services. Customers would like to use QoS characteristics of Web
services for selecting from among several alternate implementations. Each of the potential service provider declares similar functionalities for the same purpose – thus the customer expects more information about services. Typical among QoS properties are security, reliability, and performance [8]. WSDL should be extended in order to provide QoS related information with services. Once non-functional properties of services are specified, it will be possible to develop or extend tools for the discovery of Web services based both on functionality and non-functional properties. Additional tools can be designed for service aggregation, integration and composition based on QoS characteristics.

As we proceed with the quality-aware extension to the specification of services, it will be necessary to define standard metrics for non-functional properties. The Cloud Council that is developing a practical guide to service level agreements [2], recommends using ISO definitions [1] for standard metrics. Service composition leading to the computation of QoS properties of the composed services present new challenges. Consider for example “response time” as a non-functional property, and consider the composition of two services with 3ms and 5ms response times. One cannot assume that the response time of the composed service is 8ms, since computation of service times are based on stochastic measures and it may become necessary to use appropriate models (e.g., queuing theory) for computing the response time of the composed service. The orchestration of services in a composed service plays an important role in modeling QoS properties of the composite service. In the case of performance, additional complexity results from the current workload at a processing node: a lightly loaded node leads to faster response times. This may necessitate specification of performance properties at different levels of workloads (e.g., at low, average and heavy loads). These complexities can be managed using ontologies for the specification of non-functional properties.

Since the motivation of this work is not only to discover of services meeting QoS requirements, but also to compose services leading to new services and ascertaining the QoS properties of composed services, this existing available QoS extensions (see Chapter 2 for a review of some of these extension) do not fully meet acquired needs. Hence designated
extensions to WSDL are proposed to specify QoS properties. To exemplify the utilization of these extensions, this research proposed a framework for service composition with the assistance of both ontological and performance modeling tools. QoS properties are modeled with the ontological engine that can be expanded in accordance with the service declaration. Properties that are subject to a chosen performance tool can also be noted in the ontology model for further semantic comparisons. In this work, we will focus on the performance aspect of the service; in particular service response times, utilization, and throughput. As a proof of concept, the following sections demonstrate the process of WSDL extension, along with its corresponding QoS ontology modeling, performance modeling, and service composition using an example.

3.2. QoS-Aware WSDL

WSDL is the standard language suggested by World Wide Web Consortium (W3C) for service specification. It can be read as a conceptual model consisting of components with attached properties, which collectively describe the service [21]. A WSDL specification contains abstract and concrete descriptions of the service. At abstract level, it describes the interface to the service: operations with message exchange patterns (MEP) and parameter types. At the concrete level, a binding specifies the transport type that the interface uses. An endpoint then associates a real network address with the binding, which forms the service. The service is invoked by supplying the declared signature to the interface through its endpoints.

Although the syntactic specifications provide information about the structure of input and output messages, and the functional descriptions of the service, WSDL does not address non-functional properties. To fully utilize Web services, non-functional information, along with functionality, is needed in the service description. To augment any proposed extensions, backward compatibility and its extension level must be considered. Since WSDL description model addresses abstract and concrete components with services, the non-functional extensions to WSDL should be considered accordingly. It should be compatible with the original Web services mechanism in that the addition may be considered optional. Web
service engines and operations should be able to freely ignore the QoS information as they choose to operate in the conventional environment. For applications that adapt the framework, the QoS-aware extensions are extracted easily. It is decided that the extensions should be established at service level rather than at interface level, since the WSDL interfaces are bounded by the message exchange patterns and considered abstract models. At service level, an endpoint is where the abstract service binds to a concrete port type, where the overall service performance can be noted.

WSDL2.0 Core standard provides element-based extensibility that can be used to specify technology-specific binding. An element in WSDL to represent QoS property specification was created, and it was used as the extension element to the endpoint. The service with the endpoint is therefore being annotated by the extended properties. For a QoS property extension element, complex type in the XML schema is used to accommodate the data structure of the QoS. As depicted in Figure 3.1, the QoS-aware extension schema exemplifies a non-functional property of performance. Within the performance criteria, response time is noted with its value, unit, and category. The extension can also be further referenced by importing latest XML schema version which can be updated on-the-fly by revising the QoS ontology model, thus conforming to the latest XML standards.

![Figure 3.1. QoS-Aware WSDL Schema for Performance Parameters](image-url)
3.3. Performance Service Composition

Service composition decisions have to be made from considerations of both functional and non-functional requirements. To manage the semantics of both aspects and facilitate the automatic selection of service components that meet the service level requirement, an ontology engine is proposed to efficiently and flexibly classify both functional and non-functional attributes. Services and their components can be further classified according to the application domain, using the category scheme, to facilitate the retrieval and management of corresponding services. The availability of desired service depends on the discovery resulting from querying the ontology model. In case no services meet the requirements, existing services can be acquired and composed. The newly composed services can then be added to the ontology engine for future selections.

In some cases, computing the non-functional properties of composed services requires stochastic models. Consider performance properties of services such as response time. These performance attributes are used to filter and rank services so that service selections can be made. As new services are composed from its constituent service components, performance indexes can be generated by modeling the new composition through the stochastic model. The performance evaluation results along with the new composed service are added back to the ontological model for future reference.

The backend of the composition framework provides interfaces in utilizing ontology model and models for the evaluation of QoS properties for Web service composition. The two modules are independent and any potentially compatible models and tools can be plugged in. In the following, the process of creating the ontological model, and use a queuing model for composition of performance related properties of services are illustrated.

3.3.1. Ontological Property Model

Ontologies offer more accurate and flexible cataloging of entities than taxonomies. While the latter uses hierarchical and branched static structures to group entities and manage information using structural organizations, ontological model annotates semantics with meta-data, relating properties and attributes with more complex organizations than branching
or tree like structures. Ontology model therefore provides more flexible organization and semantic interpretation of data with entities.

The quality of service property of a Web service can be inferred by its performance attributes. Criterion of service selection can be formulated by the configuration of these attributes to indicate levels of service importance. Due to the dynamic nature provided by service oriented mechanism, even the meaning of performance metrics should be adapted to fit the context of the service domain. For example, a Web service component qualified for soft real-time application may be considered if they have reasonable response times; however they may not be suitable for hard real-time environments, unless the response times can be bounded. Different contexts impose different semantic interpretations on the same non-functional properties. However the ontology model is highly flexible and thus multiple semantic interpretations can be associated with properties associated with services.

To create an ontological model for Web services, leading to service composition, the process of establishing performance as non-functional property of Web services, such as response time, server utilization, and throughput are demonstrated. Further context related performance indicators can also be easily added with similar considerations. In order to create, update, and query the performance properties during the Web service composition process, we need to establish records of each and every service. This work adapts the Protege Editor [47] as the editing tool to help create an ontology model. Protege Editor has a GUI interface [37]. Users can specifically define Entity and Class as first-class elements in the schema along with their Object Properties. Instance of object can be initialized as an Individual and its Data Property can be appended. Visual tools are provided by the editor’s plug-ins to facilitate various aspects views. There are also several reasoners available that can be invoked to check and infer the ontological derivations automatically.

The first-class elements from base performance and model relevant ones are differentiated. The model refers to the performance modeling used in the compostion. The base properties serve as the mandatory performance attributes that all the Web services are required to specify as performance indicators. The model-related properties serve as the sup-
plement to the application-specific modeling approaches, thus can store additional attributes for use by specific methods and tools. In our example, the base performance classification is represented by Quality-of-Services (QoS). The QoS subclasses ResponseTime, Throughput, and Utilization are base performance indicators, for performance property. Model-related attributes include Workload and Statistics. Workload here is used as an attribute to evaluate the significance of base QoS properties. The attributes allow for recording the criteria under which the performance properties were derived and thus allow for adjustments when new running environments differ from these values.

For each of the base and model-related first-class elements, classification can also be refined into detailed subclasses. For instance, a response time can be ranked into subcategory such as Fast, Quick, Normal, Slow and Sluggish. Each of the rank can also be noted with values that represents the class based on specific context. As the new service composition emerges, the new service can easily be accommodated in the ontological model, establish quantity and corresponding semantics, and is ready for further queries and reasoning. The example model described here includes base and model-related classes and depicted in Figure 3.2. The refined QoS rank subclasses example is depicted in Figure 3.3.

To be able to interface with Protege Editor so that users can update and query the ontology model for service composition processing, the ontological process was further converted into programming. Protege-OWL provides the capability to convert API to equivalent
GUI functions and the mechanism for plugging reasoners [46]. The process was made programmable by following the process steps of creating the ontology model. The base QoS schema can serve as the building block for the extension of the ontology model. The automation enhances the flexibility to experiment on the first-class cataloging and their refined properties. It also provides a convenient facility to plugin a specific model-related ontology for performance modeling.

The automation process can be further extended with a reasoner to the ontology model that enhances the reasoning ability while interacting with the model. A reasoner implementing the reasoner plugin programming interface will be accessible in the same way that the built-in reasoners are. Jena Framework from Apache was chosen as the reasoner mechanism for the running example here. It is widely used and an open source tool. Its query and storage architecture can enable more flexible online usage of the ontology engine.

Jena programming API [43] was adapted to read the ontology model built from Protege-OWL, and validate the model by the reasoner rules. For inference support, Jena provides a general purpose rule engine that the ontology model can be validated and the application specific rule can be applied to facilitate aspects of Web service composition management. The service composition can be fine-tuned by using the rules from domain experts.
or engineers that impose application related restrictions. For instance, assume the response time of a quick Web service is defined to be less than five milliseconds, the selection of the candidate Web service can be filtered by the rule:

\[\text{print-a-quick-WebService: } \langle x \text{ pre-ws:hasQoS } a \rangle \ (a \text{ pre-ws:hasResponseTime } b) \\
(b \text{ pre-ws:rt\_value } c) \text{ lessThan}(c, 5.0) \rightarrow \text{print}(x, \ 'has\ quick\ QoS:', c) \]

The print-a-quick-WebService rule prints out any service entity that has a QoS property with a ResponseTime value smaller than 5.0 msec. Similarly, other plug-in rules can be used to customize the model to meet the needs of an application, such as performance rank selection in a service category. It should be noted that it is possible to define a reasoner that uses context related information to define fast, slow response times subjectively, instead of using values.

The ontology model automation enables the Web service composition to be processed online. New classes and properties can be created on-the-fly to address the specific needs of applications. Web service processes can also benefit from adaptation to different service domains by interpreting the performance parameters. The online feedback from the analysis is the up-to-date data that enhances accuracy of the reasoning.

3.3.2. Performance Modeling for Service Composition

Different methodologies for evaluating performance of software services such as process algebra, queueing networks, and Petri nets come with different analysis tools (e.g. PEPA [35], LQN [33], and SPNP [22]). Stochastic performance models have been widely used in the performance evaluation community. In the Web services community, it also plays an important role in assuring that the service performance meets service level agreements.

The purpose of our framework is to provide a platform that enables the use of appropriate tools for performance evaluation in Web service composition. According to the approaches the process takes, developers can explore different tools to fit the nature of the composition. Appropriateness can also be explored by comparing various tools for their usability. To demonstrate the usability of the framework, the following explains the use of a queueing model with services containing mandatory performance attributes.
While composing services, the flow among the component services can be described using a workflow or business logic. Each of the services can be represented as service nodes, and the request flow can be modeled as waiting queues. In front of each service node, requests are waiting in line for the service to process them in order. The composition model is formed with the integration of the coordinated services network. The performance outcome of the queueing network is the performance result of the newly composed service. The mapping is seen as a close fit to both the performance evaluation mechanism and the Web service composition concept. In the example of our case study, layered queueing model [33] is adapted as the tool to demonstrate our framework.

Layered queueing model is a conventional queueing model embedded with the architecture of a software system and needed resources [33]. The first class elements are processor, task, entity, and activity. A task represents a resource that has processors and other entities to execute. Each of the entities in turn can invoke other entities on other tasks to fulfill a job. These invocations are modeled in layered fashion, and can be depicted as a directed graph. For further detailed modeling, each entity can be represented with more specific activities in its own data flow. For each task and activity, there will be resource requirements specified as service time that denotes a performance attribute. And, the mean number of calls represents the average of invocations from one entity to another. With the information for each task and their entities noted, a queuing network can be constructed to represent the integration of all the tasks, leading to workload model using either open or closed queuing models. The former can be modeled with mean arrival and service rates, while the latter can specified using mean value analysis (MVA). As soon as the model is developed, the layered queueing solver can generate reports on the performance indexes such as service time, throughput and utilization for both services and processors. It also generates average waiting times in open queuing model and mean delay in closed models.

The simplest form of a Web service composition involves two services, say WS_a and WS_b. The possible compositions of the two services can be sequential or parallel composition, say WS_rs and WS_rp. Borrowing the syntax from generic process algebra, the
sequential composition can be represented as $\text{WS}_r=\text{WS}_a.\text{WS}_b$, and the parallel composition can be represented as $\text{WS}_r=\text{WS}_a||\text{WS}_b$. Assume $\text{WS}_a$ and $\text{WS}_b$ each represents an entity in different tasks say Task\_a and Task\_b. Each task is assigned to run on its own processor on different hosts say Proc\_a and Proc\_b. The sequential and parallel composition examples with an open arrival rate 0.5 are depicted in Figure 3.4. Note that the arrival rate is categorized as workload in the ontology model, and the example just serves as an instance. In the case of similar services encountering same workload but running on different platforms, the selection process have to compare the performance indices such as response time or throughput.

The service composition in both sequential and parallel topology can be scaled by accommodating multiple services at once. Resources can be exclusively owned or shared among services. Service composition can be based on either serial or parallel composition of the services involved. The final layout of the queueing network is the conceptual modeling of the Web service composition. The model can be solved by the analyzer and generate the performance indexes for the composed service.

3.3.3. Compositional Semantic Web Services

To export the ontological result that is acquired by the Web service composition mechanism, Axis [6] was used as the service publishing interface for demonstration purposes.
The interface also enables the abstraction that Web service ontological engine (WSOE) and performance modeling engine provide.

The WSOE provides for composition of services in the context of Web services management. The utility of the composition services include basic service information maintenance and composition. Service management functions include insertion, update, and deletion. WSOE_Insertion creates a record in the performance ontology model with its name and associated performance properties. The performance property in our running example is the response time of the service. Other non-functional or QoS properties can also be included within our framework. WSOE_Update and WSOE_deletion are used to update and remove the correspondent services.

New service composition information created by the WSOE can be obtained by the WSOE_Compose_Seq or WSOE_Compose_Par. The former will take the list of Web services in the order specified, and model them as a sequential network in the layered queueing model. The output will be the performance indexes for the composed service. For our simple example, the composition would return the predicted response time. Likewise, WSOE_Compose_Par will take a list of Web service in the argument, and model them as parallel network in the queueing model. The sequential and parallel compositions can be combined to obtain any general compositions of services. The list of the service interfaces are listed in Table 3.1.
3.4. Case Study

To demonstrate the Web services composition framework, a facial recognition service is used as an example. This example was chosen because of our familiarity with it while working on a related project on service composition. The service collects image data from an attached camera and identifies the presence of human faces. The service consists of facial detection (FD), image converter (IC), and facial recognizer (FR), in that order. First each of the component services are described using our QoS-aware WSDL to denote both functional and non-functional properties. For each of these services we keep their QoS records in the ontology repository. We will assume that the services are all registered so that search engine can match potential candidates meeting both functional and non-functional requirements of the customer.

To create the composed service, a list of qualified candidates of each component services are evaluated. Let us assume that our selection picked FD_E, IC_E, and FR_E as service components. We will now evaluate the non-functional values for response time of the composed service. We model the layered queueing network as follows. The service components are mapped as the entities in the layered queueing network with their correspondent tasks PD_T, IC_T, and FR_T, each of which uses processors FD_P, IC_P, and FR_P. The modeling script and the performance indexes of the example are shown in Table 3.2.

Furthermore, let us assume that image converter (IC) service can be composed in parallel to improve performance. The composition engine can be configured to explore the service composition using parallel workflow among the services. The new composition would use two image converter (IC) services in parallel named IC_E1 and IC_E2. The modeling script of the example and the performance indexes result are shown in Table 3.3.

Although only a simple example and a single property is used here, the proposed framework is very general and flexible so that it can be easily extended for more complex service discovery based on many QoS properties, and can be composed in very complex manner.
### Table 3.2. Layered Queueing Model for Facial Recognition Service Composition Example

<table>
<thead>
<tr>
<th># General Section</th>
<th>Service time:</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Task Name Entry Name Phase 1</td>
</tr>
<tr>
<td>&quot;Web service modeling.&quot;</td>
<td>User_T User_E 50.4902</td>
</tr>
<tr>
<td>0.00001</td>
<td>FD_T FD_E 0.5</td>
</tr>
<tr>
<td>1</td>
<td>IC_T IC_E 0.4</td>
</tr>
<tr>
<td>0.9</td>
<td>FE_T FR_E 0.3</td>
</tr>
</tbody>
</table>

### Processor Information

<table>
<thead>
<tr>
<th>P 0</th>
<th>p User_P f i</th>
<th>p FD_P f</th>
<th>p IC_P f</th>
<th>p FR_P f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P f i</td>
<td>FD_P f</td>
<td>IC_P f</td>
<td>FR_P f</td>
</tr>
</tbody>
</table>

### Task Information

<table>
<thead>
<tr>
<th>T 0</th>
<th>t User_T User_E m 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t FD_T FD_E FD_P</td>
</tr>
<tr>
<td></td>
<td>t IC_T IC_E IC_P</td>
</tr>
<tr>
<td></td>
<td>t FE_T FE_P FE_P</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

### Entry Information

<table>
<thead>
<tr>
<th>E 0</th>
<th>s User_E 0 -1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y User_E FD_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s FD_E 0.1 -1</td>
</tr>
<tr>
<td></td>
<td>y FD_E IC_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s IC_E 0.1 -1</td>
</tr>
<tr>
<td></td>
<td>y IC_E FR_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s FR_E 0.3 -1</td>
</tr>
</tbody>
</table>

### Table 3.3. Layered Queueing Model for Facial Recognition Service Composition Example

<table>
<thead>
<tr>
<th># General Section</th>
<th>Service times:</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Task Name Entry Name Phase 1</td>
</tr>
<tr>
<td>&quot;Web service modeling.&quot;</td>
<td>User_T User_E 82.342</td>
</tr>
<tr>
<td>0.00001</td>
<td>FD_T FD_E 0.82</td>
</tr>
<tr>
<td>1</td>
<td>IC_E2 0.36</td>
</tr>
<tr>
<td>0.9</td>
<td>FE_T FR_E 0.3</td>
</tr>
</tbody>
</table>

### Processor Information

<table>
<thead>
<tr>
<th>P 0</th>
<th>p User_P f i</th>
<th>p FD_P f</th>
<th>p IC_P f</th>
<th>p FR_P f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P f i</td>
<td>FD_P f</td>
<td>IC_P f</td>
<td>FR_P f</td>
</tr>
</tbody>
</table>

### Task Information

<table>
<thead>
<tr>
<th>T 0</th>
<th>t User_T User_E m 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t FD_T FD_E FD_P</td>
</tr>
<tr>
<td></td>
<td>t IC_T IC_E IC_P</td>
</tr>
<tr>
<td></td>
<td>t FE_T FE_P FE_P</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

### Entry Information

<table>
<thead>
<tr>
<th>E 0</th>
<th>s User_E 0 -1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y User_E FD_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s FD_E 0.1 -1</td>
</tr>
<tr>
<td></td>
<td>y FD_E IC_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s IC_E 0.06 -1</td>
</tr>
<tr>
<td></td>
<td>y IC_E FR_E 1 -1</td>
</tr>
<tr>
<td></td>
<td>s FR_E 0.3 -1</td>
</tr>
</tbody>
</table>

Throughputs and utilizations per phase:

Total: 2.42889 0.874401 0.874401
4.1. Introduction

Service oriented architecture (SOA) is a flexible and scalable design methodology to seamlessly integrate and cooperate services in distributed software and systems. As more services are on the web and in the cloud, it becomes easier to create new services dynamically by composing existing services, customized to meet the needs of customers [32]. Before invoking a service, a service requester has to query the functionality as well as the interaction protocols defined to access the service. Web Service Description Language (WSDL) [21] is a widely accepted standard from World Wide Web Consortium (W3C) for describing functionality of Web services. The Universal Description, Discovery and Integration (UDDI) registry serves as a repository for the services with WSDL descriptions. Users can query the UDDI and find services meeting their needs since the functionality of the services can be obtained from their WSDL specifications [27].

Once the services are selected, interactions among the services are achieved using messaging protocol defined in WSDL. Even with ever increasing number of services, it may still not be possible to find the "right" service, and in such cases, one has to either create a new service from scratch, or compose the service using existing services. Tools and frameworks are becoming available to aid in the dynamic composition of services [87, 55, 59, 36, 14]. Another issue that needs to be addressed is related to selecting the appropriate services that takes part in a composition, particularly when multiple services with the same functionality are available. In such cases, non-functional or quality of service (QoS) properties, such as performance, security, reliability become the delimiters [51, 49, 69, 82].

While standard WSDL describes the functionality of a service, it does not specify QoS or non-functional properties. As described in previous chapter, the WSDL was augmented to permit specification of non-functional properties of a service. The additional information can help distinguish between services with the same functionality, and these properties can be
used while composing new services to ascertain the QoS properties of the composed service.

Enterprise software systems or Cloud computing often use business logic to refine their design and regulate the behavior of services according to business processes [4]. Business Process Execution Language (BPEL) has become the standard for describing the architecture of a service process [28]. It contains control constructs for the orchestration of component services in a workflow style. While tools and frameworks are available to use BPEL orchestrations in composing services, they are not suitable to evaluate the QoS properties of specific orchestrations [61, 52]. In this chapter, the proposed framework is expanded to adapt the notion of BPEL to describe QoS-aware services for their selection and composition. As have shown in the previous chapter, BPEL is compatible for use of QoS extensions. The expansion is also backward compatible with the SOA in general and Web services in particular. It is suitable for the incorporation of any tools that facilitate QoS extensions and models for analyzing QoS properties. This chapter illustrates how to create queuing models for various BPEL orchestration logic compositions. In the case study, the extended framework is demonstrated for composing performance properties using stochastic models, such as the Layered Queueing Network (LQN) model.

4.2. QoS-Aware Framework in SOA

In this section, the QoS-aware framework in the SOA environment that extends from the framework presented in chapter 3 is presented. Firstly, essential components of the framework, and its operational processes are described. Then, the quality awareness extension to the business processes are considered, specifically using WS-BPEL [45] to enable the performance evaluation result in service design, selection, and composition which include operational logic choices.

4.2.1. Framework Description

As shown in previous chapter, WSDL is extended to permit specification of non-functional property elements with services. Each Web service can optionally describe QoS properties along with functional properties, in order to distinguish itself from other services
providing similar functionality. QoS properties can include performance, reliability, security or other quality metrics. The framework is compatible with traditional SOA for either standard or quality-aware service publication, selection and interaction. The infrastructure of the framework augments the SOA with three elements:

- **QoS-Aware WSDL Extension (QoS-WSDL):** These are new WSDL elements for specifying QoS properties with services.
- **Ontological QoS Modeling (QoS-Ontology):** QoS properties and categories are classified by our ontology model allowing different classes of QoS properties and relationships among these categories.
- **Testing and Composition QoS Modeling (QoS-TestCompose):** QoS properties can be used for selecting services and evaluating QoS properties of composed services using component properties. The composition properties can be evaluated based on different ontological classes and relationships.

The QoS-aware framework for SOA is depicted in Figure 4.1. For a QoS-aware service to work, QoS-WSDL has to be prepared by instantiation of the QoS-Ontology model of the
service. The value of the QoS properties can be obtained by using QoS-TestCompose as a testbed for analytical modeling or testing (P1). Once the Service Provider equips the service with the QoS extension, the provider can register the service specification on the Service Registry (P2). A Service Requester can query the Service Registry to discover qualified candidates by examining both functional and non-functional properties (D1). Functional properties are interpreted by reading WSDL, while non-functional properties are referenced by the QoS extensions. In case multiple services are selected with the same functional properties, the requester can use QoS properties to differentiate between the services. The selected services can be used to create new composed services and the QoS properties of the new services be obtained using the QoS-TestCompose Modeling (D2).

For those services that have not used our QoS extensions, our framework uses conventional selection and composition processes (P2 and D1).

4.2.2. QoS-Aware Framework with WS-BPEL Extension

The QoS-Aware framework are extended to use business processes. Service declaration types with atomic and process descriptions are discerned. An atomic service (AS) is the one whose provider offers functionality with design details but implementations hidden. Access to the service is achieved with required message exchange pattern (MEP) and binding of ports as shown in WSDL. In other words, an atomic service is opaque and represents the standard Web services. The previous work illustrated service selection and compositions with respect to service performance in [50]. However, at the time, it did not use any specific logic for the composition.

A process service (PS) is a service that facilitates collaborations between services controlled by business logic. A PS may be composed by multiple AS and/or PS. PSes can be nested. At the lowest layer of the hierarchy, a PS should only consist of ASes. It will show in the following that both AS and PS can be modeled and analyzed in our QoS-aware framework.

There are many well-known business process modeling languages available to formally describe the interactions among different service components with business logic [56]. These
languages rely on well-defined workflow formats. In some cases they use meta-data that can be used for management purposes. In this work, we use WS-BPEL or BPEL for short, to demonstrate the service selection and composition capabilities of our framework.

In the Web service context, BPEL can be treated as a layer on top of WSDL [67]. BPEL provides the description of behavior and interactions of a process instance with its partners and resources through Web Service interfaces. Both the process and its partners are exposed as WSDL services [45]. Furthermore, BPEL follows WSDL model of separation between abstract information, such as message and port type, and concrete information, such as binding and endpoint. The two use cases for modeling BPEL processes are abstract and executable. Abstract processes describe the protocol that specifies the message exchange between parties without revealing their underlining implementations. While abstract processes may hide some of the required operational details, executable processes are fully specified and can be executed. Both abstract and executable processes share all the BPEL constructs, except that the former has additional opaque mechanisms for hiding operational details [45].

To include a PS in the framework, we assume that WSDL descriptions for all services are available. WSDL files describe how to use services, while BPEL describe collaborations among the services or tasks. In accordance to our design of the QoS-aware framework, only concrete WSDL is relied upon in our QoS-aware framework. Quality of Services with concrete bindings provides more specific range of values, derived from actual tests or analyses. The service that is extended for use in our framework can be viewed as an AS with a concrete WSDL. Or an abstract AS can be included in our framework, provided the QoS properties are derived through a concrete binding (as shown by process P1 in Figure 4.1).

Now we consider if the assumptions can be applied to the cases of acquiring a PS in the framework. For an executable PS, it is natural to assume that the services involved in the PS have concrete WSDLs, since an executable process is assumed to be concrete. For an abstract PS to be included in the framework, it must first be transformed into an executable PS. The transformation is called Executable Completion [45] in the Web services context of WS-BPEL. The main algorithm of the transformation and related issues concerning QoS
properties will be addressed in later sections.

With the adaptation of PS into the framework, we now consider the process of publishing and discovery operations for processes. Once again, an executable PS can be observed as services with concrete WSDL in the framework. To publish a PS service, it applies the same process P1 and P2, shown in Figure 4.1; for service registration of ASes, additional service meta data is added to the QoS_Ontology compartment for describing management related information. Since the framework differentiates a PS from an AS, the ontology model notes the service identification and service type classification when a service is instantiated. The additional information includes identification of a PS, the business process structure, and its sub-components. Note that the additional information of a PS is stored in the framework and is independent of the data minted in a Service Register of the SOA triangle. To discover a PS service, it makes no difference as to discovering any AS with QoS annotations in its registered WSDL (D1). Re-discovery of a PS service is required to first discover its sub-services as ASes, and submit the business process to QoS_TestCompose for updating QoS values (D2).

Although only concrete AS and executable PS are allowed in the framework, abstract processes can still be included. An abstract process can be viewed as embedding multiple use cases. The use cases are differentiated by their usage profiles. From the abstract processes, one can analyze the profiles to obtain specific values for QoS properties of the processes. To this end, records of abstract processes should be kept so as to facilitate QoS-aware compositions using different business process operations. The reasoning of how PSes can be used in our framework is discussed in Section 4.3.

4.3. Service Compositions with WS-BPEL in the QoS-Aware Framework

SOA enables a flexible and adaptable Web service discovery and service composition. To allow for selection and composition based on QoS properties within our framework, we need to devise processes to guide QoS-aware business process selection and compositions. Since WS-BPEL is an established standard to describe business processes in the Web services context, BPEL is used to describe business processes in the framework.
Orchestration and choreography are two aspects of creating businesses from composite Web services [67]. Orchestration refers to an executable process that interacts with internal and external Web services. Since the executable process may include business logic and task execution order, it represents the control flow among the participating services. On the other hand, choreography refers to the interactions (or data flow) among participants who cooperate to achieve the objectives of the composed services. Choreography coordinates message exchanges occurring among services. For our purpose, BPEL4Chor [29], an extension of BPEL, was adapted as the choreography framework. Engineers can use the language and available tools to readily model service interactions. The following will show how this BPEL choreography can be used within the QoS-aware service composition framework.

The following subsections include discussions of the applicability of quality awareness to both orchestration and choreography compositions, and their operational processes. Note that the focus of the service composition here concerns non-functional properties while assuming the functional semantics in the selection and composition has already been completed. We use the term service candidates to refer to the services already selected for composition based on functional requirements.

Our framework is designed to permit the use of many different approaches for specifying QoS properties, provided appropriate tools for selecting services meeting specific non-functional properties are also available.

4.3.1. Business Process Service Orchestration

A service orchestration is to organize the sub-services of a PS, and the message exchange with other services to achieve its service purposes. The PSes considered here are executable with their sub-services are also executable. Since the PS and its component services are all executable, they are eligible to apply the QoS extension when registering the service in the framework. Service composition from the perspective of orchestration involves sub-services selection. The PS selection for orchestration comes down to two scenarios: a fixed process organization, and process candidates of the same functionality with alternative design.
A PS may consist of m sub-services whose organization is based on the business process logic and how the tasks are ordered. Candidates for the m component sub-services are selected based on both functional and non-functional (QoS) properties. Since candidate services are assumed to use QoS extended WSDL, QoS references can be obtained for services and appropriate service components can be selected based on QoS properties.

In the case of multiple candidates of the same services with alternative business processes, they can be further classified as fixed or non-fixed sub-services. If the sub-services are fixed, the whole PS can be treated as AS. Then, the service selection only involves comparing the QoS values of the targeted non-functional attributes.

If the sub-services can be changed dynamically, each of the process candidates may be evaluated using multiple use cases. Each use case that belongs to a process candidate must be re-discovered for its QoS values. The result of QoS criteria for each candidate can be obtained, which can be used to match the requirements in order to make the decision.

4.3.2. Business Process Service Choreography

As stated perviously, choreography describes the interaction protocols (or data flows) among component services of a business process. While orchestration utilizes executable processes for modeling, choreography uses abstract process to describe the collaboration among service partners.

Since our QoS-aware framework requires concrete services with binding so that non-functional properties can be measured, the abstract nature of choreography in describing service interactions is not a direct fit for our framework. Thus, we need to extend the abstract interactions with appropriate concrete annotations of QoS attributes. Since our framework adapts BPEL as the descriptive language for business processes, we adapt BPEL4Chor [29] to model service choreography. We further annotate the interactions to make the choreography QoS-aware.

BPEL4Chor consists of three artifact types:

- Participant behavior description (PBD): It defines control flow dependencies between activities. It uses the Abstract Process Profile to describe requirements on
the behavior of a participant. The profile inherits from Abstract Process Profile for Observable Behavior specified in BPEL, with the addition of identifying activities with unique identifiers. The PBD is essentially an abstract process with the additional attributes kept in the profile.

- **Participant topology:** It defines the collaboration structure of participant types, participants, and message links. The topology describes the communication structure of the service interactions among participants.

- **Participant grounding:** It defines the actual configuration of the choreography, and shows the connections to the concrete WSDL of the service participants. For each message link defined in the participant topology, a port type and its operation is specified. After the grounding, every PBD of the service can be transformed to an executable BPEL process based on their profiles.

An initial high level mapping from the modules of BPEL4Chor to our framework is straightforward. Although our framework requires concrete service data, abstract process is included in the framework. And, it is feasible to use abstract processes during the composition process before the new grounding of composition is admitted in the framework. The processes of adapting the composition to our QoS framework are presented below. We will refer to the processes shown in Figure 4.1 in our discussions below.

- **From BPEL4Chor to QoS-Aware Extension**

  The main product of a service choreography is an executable process. The new service can be included in the QoS-extension framework by first submitting to the QoS_TestComposite for QoS evaluation (D2). Corresponding process data is established with additional specific records for a choreography including PBD for all the participants and the composition topology. Recording a PBD is compatible with storing an abstract process, which is supported in the QoS-Aware extension.

- **From QoS-Aware Extension to BPEL4Chor**

  The main activity of BPEL4Chor is to identify a set of service participants to create a new service. The process involves selecting the service participants, extracting
the PDB, and applying BPEL4Chor processes to compose the new service. The participants are restricted to only PSes since we have to identify the names of the operations. The QoS-aware framework facilitates the selection process by providing QoS values during discovery (D1). The selection process is similar to the selection process of a PS as introduced in Section 4.3.1. Once we select the participants for composition, we will need the PDB for each participant. Since each participant selected is executable PS, there always exits one and only one abstract PS in the framework that belongs to the PS. Transforming a PS to a PDB is straightforward with adding the unique name to the message exchange operations. With the named message links, practitioners then put together the required participant topology with the design. The grounding information for each linked operation is already available with an executable PS. Then BPEL4Chor composition process is complete, and the new composite service is created. To accommodate the created PS in the framework, the same processes mentioned in Section 4.2.2 is followed.

4.4. From WS-BPEL to LQN

In this section, the QoS_TestCompose module in the framework is explained to illustrate how QoS properties of services that are composed using BPEL are calculated. Modeling non-functional properties for services and their composition is not always straightforward, since combining QoS properties of different services are based on underlying mathematical models. For example, given a process with the service components executed in sequential order, one may assume that the response time of the combined process is the sum of the response times of the component services. However, this will not be accurate because combining performance properties rely on stochastic processes. In other words, for obtaining performance attributes of a composed process we must use stochastic models. In our case, the layered queueing network (LQN) is used for modeling performance. However, other stochastic models and tools can be used to compute QoS properties of processes using BPEL.

The following subsections gives a brief introduction to the essential elements of BPEL and LQN. The transformation rules are derived for mapping from BPEL compositions to
models in LQN, and how LQN can be used to compute performance attributes are discussed.

4.4.1. WS-BPEL Constructs

WS-BPEL [45] is a standard language intended to describe business processes for Web services. The idea is to represent collaborations among services or tasks described in the WSDL language. As a descriptive language using XML format to describe workflow of business process, BPEL consists of two types of activities: Basic and Structured.

Basic activities are atomic activities mainly describing service interactions. They include <receive> and <reply>, which represent waiting for a message, and response to a message respectively; <invoke> enables a Web service operations offered by a service partner. The invocation enables either a one-way or request-response message exchanges. Other basic activities include <assign> to update a value of a variable, <exit> to end the process, <wait> to delay the execution, and <empty> to express no-op operation. Still others include <scope>, <throw>, <compensate>, and <validate> that handle from the execution scopes to fault handling operations. New activity creation is also possible through <extensionActivity>.

Structured activities control the process order of activities. They can be nested in other structured activities as well. The constructs include <sequence> and <flow> to express sequential and parallel order of the enclosed tasks. The control flow constructs include <if> that sets a boolean condition for activities, <while> and <repeatUntil> that iterate through their enclosed processes until the condition becomes false; <forEach> controls the number of times the set of enclosed tasks can repeat, either running in sequential or parallel, while <pick> chooses among tasks to be executed depending on the occurrence of the event.

4.4.2. Layered Queueing Networks

Layered queueing network [34] is an extended queueing model with the layered structure representing servers at higher levels making requests to servers at lower levels. Each task in the model involves sharing and consuming processing resources. An entry of a task can be modeled as the service operation stub receiving requests and responding with a reply
to higher level systems. The entry can be further refined with activities representing the workflow of its sub-components which are organized with precedence operators, such as fork and join. For each task and activities, there are resource requirements specified, as service demand in time. The interactions between different servers and their tasks can be modeled with phases representing message receipt and response in different time slots. The nature of the communication can be defined as synchronous and asynchronous, which model blocking and non-blocking interactions respectively.

As modelers put together the service architecture and information needed for the system integration, a queueing network is created. The system modeling can be subjected to either open or close networks during performance analysis. LQN comes with an analytic solver (lqns) and a simulator (lqsim) to generate the performance indexes such as response time, utilization, and throughput.

LQN models can also be expressed in XML format. A further analysis to explore the design space with different combination of system configuration is also possible with its LQX tool. LQX is a general purpose programming language used for the control of input parameters to the LQN solver system. The language allows a user to put together a wide range of different set of input parameters, and solve the model accordingly.

4.4.3. Transition Rules from BPEL to LQN

A structure of business process in BPEL largely consists of activities and their corresponding fault handlers, in addition to variables, correlation sets, and partner links. Since performance evaluation of the business processes is the focus here, the derivation of the transformation rules only focus on the process activities. For the performance analysis purpose, the activities in the event and fault handlers can follow the same set of rules, and integrated with the activities in the main processes.

The main process activities usually begins with a list of sequential activities. The behavior of the activities, both basic and structured, are described by the control constructs. The main task of the transformation is to maintain the same activity orders as in BPEL when creating the LQN model. For basic activities, the order of the behavior relates to
mainly communication protocols. For structured activities, the order can be focused on the mapping of business logic.

The order of the control flow in the transformation is realized using precedence of activity connections in LQN tasks. The precedence can be sub-classed into Join and Fork for modeling synchronization and concurrency of activities. To connect one activity to another, the source activity connects to a pre-precedence (or Join). A pre-precedence in turn connects to a post-precedence (or Fork), and then to the destination activity. More details on precedence types can be found in the LQN User manual [34].

Service requests in LQN can be of three types: rendezvous, send-no-reply, and forwarded. Rendezvous is a blocking synchronous request, while the send-no-reply is an asynchronous request. Forwarded requests are redirected to a subsequent server, which may forward the requests again, or reply to the original client. In the translation, we consider the message exchange pattern to match either blocking or non-blocking, and either one-way or two-way for service invocation.

The summary of mapping of basic constructs are listed in Table 4.1, while the mapping of structured constructs are listed in Table 4.2. For each mapping entry, a brief description is included. For those elements that have no direct LQN semantic counterparts, we use (N/A) with explanation. Since the focus of the transformation is on performance analysis, the corresponding performance models for fault handling activities should be obtained by following the error handling mechanisms designated in the processes. The handling processes can then be subjected to the transformation rules to obtain appropriate performance models. The part of fault handling of the transformation and its performance evaluation is not included in this work.

4.4.4. Data Dependency in Transformation

There is no direct equivalent LQN transformation for the BPEL conditional construct such as if-else. However, an Or-Fork representing a branching point with a given probability p to a selected process path can emulate the semantics of if construct. The probability is set to 1.0 for the if-clause, if the condition should be evaluated to true. On the other hand,
Table 4.1. Mapping BPEL Basic Activity to LQN Elements

<table>
<thead>
<tr>
<th>BPEL Basic Activity</th>
<th>LQN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;receive&gt;</td>
<td>Pre-precedence (or a join-list)</td>
<td>Getting a message from a service partner.</td>
</tr>
<tr>
<td>&lt;reply&gt;</td>
<td>Request (send-no-reply) : direct reply Request (forwarded) : indirect reply</td>
<td>Sending a message to a service partner.</td>
</tr>
<tr>
<td>&lt;invoke&gt;</td>
<td>Request (send-no-reply) : one-way Request (rendezvous) : request-response</td>
<td>Invocation of a service offered by a service partner. It can be one-way or request-response interactions.</td>
</tr>
<tr>
<td>&lt;wait&gt;</td>
<td>Activity with a think time</td>
<td>A delay for a timer.</td>
</tr>
<tr>
<td>&lt;empty&gt;</td>
<td>Activity with zero service time</td>
<td>A no-op holder which does nothing.</td>
</tr>
<tr>
<td>&lt;exit&gt;</td>
<td>N/A</td>
<td>Immediate termination</td>
</tr>
<tr>
<td>&lt;assign&gt;</td>
<td>N/A</td>
<td>Assign a value to a variable.</td>
</tr>
<tr>
<td>&lt;validate&gt;</td>
<td>N/A</td>
<td>Validate the value of variable defined in WSDL.</td>
</tr>
<tr>
<td>&lt;throw&gt;</td>
<td>N/A</td>
<td>Generate a fault from business process. Fault handler needs to be specifically modeled.</td>
</tr>
<tr>
<td>&lt;rethrow&gt;</td>
<td>N/A</td>
<td>Regenerate a fault from fault handler. Fault handler needs to be specifically modeled.</td>
</tr>
<tr>
<td>&lt;compensate&gt;</td>
<td>N/A</td>
<td>Compensate actions can not be completed. Fault handler needs to be specifically modeled.</td>
</tr>
<tr>
<td>&lt;compensate-Scope&gt;</td>
<td>N/A</td>
<td>Compensate actions can not be completed in a specified scope. Fault handler needs to be specifically modeled.</td>
</tr>
</tbody>
</table>

the else-clause will be taken with the probability of the if-clause set to zero, if the condition should be evaluated to false. The transformation from <if> in BPEL to LQN can thus be expressed using the semantic of Or-Fork and Or-Join with appropriate probability p. The probability depends on the variables involved in the condition. The frequency of which path is taken depends on the statistical or empirical evaluations. Each sample represents a specific service system configuration that is invoked in a specific use case.

The conditional variables can be related to either service workload or the frequency of the variable assignments. For example, in a sorting algorithm, workload, viewed as the input size, can impact the service time. The business process may consider splitting the input into smaller sizes, and merging the result later. The condition may also depend on a multivariate function when multiple outcomes are possible. Data profiling and other
Table 4.2. Mapping BPEL Structured Activity to LQN Elements

<table>
<thead>
<tr>
<th>BPEL Structure Activity</th>
<th>LQN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sequence&gt;</td>
<td>Precedence: Sequence</td>
<td>A list of service activities executed in the specific order.</td>
</tr>
<tr>
<td>&lt;flow&gt;</td>
<td>Precedence: And-Fork &amp; And-Join</td>
<td>A bag of service activities executed in concurrent and finished in synchronization.</td>
</tr>
<tr>
<td>&lt;if&gt;</td>
<td>N/A [Use Or-Fork &amp; Or-Join to emulate the condition with a probability 1 or 0.]</td>
<td>Take different actions depends on the Boolean condition.</td>
</tr>
<tr>
<td>&lt;pick&gt;</td>
<td>N/A [Use Or-Fork &amp; Or-Join to emulate the condition with a probability p.]</td>
<td>Activity is chosen depending on the kind of message or timeout events.</td>
</tr>
<tr>
<td>&lt;while&gt;</td>
<td>N/A [Precedence: Loop to emulate the number of iteration.]</td>
<td>Iteration on the Boolean condition evaluated to true.</td>
</tr>
<tr>
<td>&lt;repeatUntil&gt;</td>
<td>N/A [Precedence: Loop to emulate the number of iteration.]</td>
<td>Iteration will stop on the Boolean condition evaluated to true.</td>
</tr>
<tr>
<td>&lt;forEach&gt;</td>
<td>N/A [Precedence: Loop to emulate the number of iteration]</td>
<td>Repeat activities multiple times, activities in each iteration can be modeled with &lt;sequence&gt;or &lt;flow&gt;</td>
</tr>
</tbody>
</table>

empirical evaluations can be used to assign probability values with each outcome [40]. Similar approach can be applied to <pick> where the Boolean condition becomes the frequency of the message variable. The sum of the probabilities of each case in the Or-Fork must be 1.

For the loop control statements, such as <while> and <repeatUntil>, the Boolean condition should be analyzed using the number of times the iteration will be executed. The counterpart in the LQN is a loop (for a service) which is executed desired number of times. <forEach> is similar to these iterative controls with the addition of specifying the execution type, in sequence or parallel, of activities in the loop clause.

4.5. Case Studies

Here we demonstrate the approaches proposed in the framework described above using a facial recognition systems. It consists of several software components, separated by their functionality, that represent service components used for Web service composition as
an example. Each business logic of a service component is presented with a corresponding BPEL description. LQN models are generated for individual and newly composed service to show the performance evaluation approach in our framework. The example also suggests the activities of data dependency and design space analysis using proposed processes.

4.5.1. Facial Detection and Recognition Example

A building security monitoring system, which uses facial detection and recognition technique, is used as the example to demonstrate how our framework can be used. The purpose of the system is to detect intruders, and raise an alarm when intruders are detected, as well as recognizing the intruders using facial recognition software to compare with existing database of stored images.

A general computation of facial detection and recognition is split into multiple tasks – signal processing, image analysis of machine learning algorithms and processes. In our example, the service is divided into three modules: Facial Detection (FD), Image Converter (IC), and Facial Recognition (FR).

- FD receives video frame input and detects if there are faces appearing in the image. If no face is detected, no action will be taken. However, if faces are detected, alarm messages will be sent and image frame will be the output for further processing.
- IC receives image frames with faces detected, and prepare the normalized file formats for each face. The output consists of the images that can be compared against images stored in the databases.
- FR receives the normalized face images as input, and sets the connections to databases containing images of faces for identification. Once there is a match, a report is sent to human operators with information about the persons identified.

The three modules will be considered as Web services, and our goal is to create a new Web service that will combine these component services, using sequential composition in the order of FD, IC, and FR. The process sequence of the three services in BPEL is shown in Figure 4.2, Figure 4.3, and Figure 4.4 respectively.
**Figure 4.2. Facial Detection BPEL (FD)**

```xml
<sequence>
<opaqueActivity name="DetectFacialProcess"/>
<if>
    <condition opaque="yes"/>
    <flow>
        <invoke wsu:id="SubmitICReq"/>
        <opaqueActivity name="SubmitAlarm"/>
    </flow>
    <else>
        <opaqueActivity name="SubmitNoResult"/>
    </else>
</if>
</sequence>
```

**Figure 4.3. Image Converter BPEL (IC)**

```xml
<sequence>
<receive wsu:id="ReceiveICReq" createInstance="yes"/>
<if>
    <opaqueActivity name="SplitImageFrame"/>
    <forEach name="splitFile" wsu:id="NormalizeFrameSize" parallel="yes">
        <startCounterValue>1</startCounterValue>
        <finalCounterValue>2</finalCounterValue>
        <scope>
            <opaqueActivity name="NormalizeMultipleImage"/>
        </scope>
    </forEach>
    <else>
        <opaqueActivity name="NormalizeNormalImage"/>
    </else>
</if>
</sequence>
```

**Figure 4.4. Facial Recognition BPEL (FR)**

```xml
<sequence>
<receive wsu:id="ReceiveFRReq" createInstance="yes"/>
<forEach wsu:id="queryDatabase" parallel="yes" opaque="yes">
    <startCounterValue>1</startCounterValue>
    <finalCounterValue>3</finalCounterValue>
    <scope>
        <opaqueActivity wsu:id="FacialRecognitionProcess"/>
    </scope>
</forEach>
</sequence>
```
Each BPEL is transformed into a LQN model for analysis. To submit the service into the framework, the LQN model is analyzed with the result of the performance indexes obtained from QoS values of individual services. The transformation of the LQN models are shown in Figure 4.5, Figure 4.6, and Figure 4.7 respectively.

The entire composition for the building security application can be sought in different ways depending on the approaches the engineers use. We demonstrate two example scenarios to show how our framework facilitates compositions. In a simplified scenario, all services can be considered as atomic services, while in a more flexible scenario, the composition utilizes the workflow processes to leverage the service choices in order to gain a better performance.

To compose the the system in the simplest case, service discovery process (D1, shown in Figure 4.1) is applied. For FD, IC, and FR, QoS values such as service execution time are obtained from their QoS extended WSDL files. A simple version of the sequential BPEL expression is created in Figure 4.9.

The transformation steps along with the quality attributes obtained from the WSDL
Figure 4.6. Image Converter LQN Model

Figure 4.7. Facial Recognition LQN Model
extension of each services, together create the LQN model of the composition. The LQN model is depicted in Figure 4.8. The new composition along with the performance indexes resulting from analyzing LQN models can be published using service publish process (P2, shown in Figure 4.1).

A more flexible way to consider the composition is to observe the Web services components as processes. We first retrieve Web services along with their processes. Applying BPEL4Chor processes, a topology file is created to build the service interactions. A snapshot of the topology configuration is shown in Figure 4.10. The result of the composition along with the derived BPEL and corresponding LQN model is shown in Figure 4.11.

4.5.2. Data Dependency Considerations

In the Image Converter (IC) BPEL process, the if-clause distinguishes between single and multiple faces that need to converted, since converting multiple faces increases workload on the processing systems. If the image contains multiple faces, it may be desirable to use multiple processes executing concurrently to improve the speed of IC process. Here we
Figure 4.10. Choreography Topology for Process Service Composition of FD, IC, and FR

```xml
<?xml version="1.0" encoding="UTF-8"?>
<topology name="example facialDetectRecognizer topology" targetNamespace="http://agentmode.com/choreography/facial/topology"
xmlns:fd="http://agentmode.com/choreography/facial/detector"
xmlns:ic="http://agentmode.com/choreography/facial/ converter"
xmlns:fr="http://agentmode.com/choreography/facial/recognizer"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<participantTypes>
  <participantType name="FD" participantBehaviorDescription="fd:detector" />
  <participantType name="IC" participantBehaviorDescription="ic:converter" />
  <participantType name="FR" participantBehaviorDescription="fr:recognizer" />
</participantTypes>

<participants>
  <participant name="detector" type="FD" selects="converter" />
  <participant name="converter" type="IC" selects="recognizer" />
  <participant name="recognizer" type="FR" />
</participants>

<messageLinks>
  <messageLink name="icRequest" sender="detector" sendActivity="SubmitICReq" receiver="converter" receiveActivity="ReceiveICReq" messageName="icRequest" />
  <messageLink name="frRequest" sender="converter" sendActivity="SubmitFRReq" receiver="recognizer" receiveActivity="ReceiveFRReq" messageName="frRequest" />
</messageLinks>
</topology>
```

model two identical servers executing the same job by splitting the conversion tasks into two assuming two faces are detected. Each server, which either processes the single task or two tasks, has the same execution performance and same capacity. The service time depends on the probabilities associated with detecting one or two faces. In this example, we vary the if-clause probability from 0.01 to 0.99, and estimate the effective performance. Figure 4.12 shows the execution time ranges while the probability with the if-clause is changed. Figure 4.13 shows the utilization of the image conversion servers.

Similar method is also be used with Facial Recognition (FR) BPEL process, using either a single or multiple tasks to compare the faces with those in the database. To further speedup the process, the database may be organized into frequently accessed faces and less
FD IC FR Choreography Model

Figure 4.11.
frequently accessed faces. In this example, we separated the facial databases into three separate databases, d1, d2, and d3. Rather than concurrently querying all three databases, modelers can select just one representative database based on the likelihood of finding a match. Figure 4.14 shows the execution time of the FR service while adjusting the probability
of success with \( d_1 \). Figure 4.15 shows the utilization versus the probabilities.

4.5.3. Performance Space Considerations

Various design topologies that yield different service performances can also be considered. In this example, system structure and server capacity are explored. Consider the IC example for multiple image conversion. Instead of running two converters concurrently, suppose we want to explore the alternatives that execute them sequentially as a two-step pipeline, where the execution time of each step is only one forth compared to the parallel ones. There are also options that the server can be equipped with single or multiple processors (e.g., multicore systems) to speed up the service. Together these options can be analyzed by the LQN. Figure 4.16 shows the service times of the converter compared to the previously shown split workflow; we use S to represent the sequential flow and P to represent parallel workflow and the suffix indicates the number of processors.
4.6. Conclusion

In this chapter, a framework for composing Web services using QoS-aware BPEL was described and how the framework can be used with a simple case study had been demonstrated. It has been showed that services in BPEL description can be seamlessly accommodated in the framework to perform service orchestration and choreography. For atomic and composite services, transformations from BPEL to LQN models for performance evaluations were demonstrated with the case study.
Developing a software system that meets its specifications demands continuous verification and validation efforts in iterative developmental cycles running from analysis and design, to implementation and deployment. During these processes, engineers build the system by creating design plans, and maintaining expected functional and non-functional properties manifested in the specifications. Testing and debugging activities take place alongside the development. Similar to conventional functional debugging, non-functional properties must also be tested and appropriate fixes made to meet the requirements. The complexity of modern software systems makes it difficult for the designer to assure compliance of non-functional requirements. New methodologies and tools that can help tracking non-functional properties of designed systems and subsystems can alleviate the problem.

A novel framework that assists in performance debugging of distributed software systems is described in this chapter. The key to the proposed framework is the use of performance anti-patterns. In order to detect performance anti-patterns, a performance baseline must be established so that the performance of the designed system can be measured for any performance deficiencies.

For each system or sub-component being evaluated, the framework creates profiles that is called suggestive profiling. When used during the development life cycle, it provides realistic means both for antipattern detections and suggested solutions during the refactoring phase of a performance debugging process. Information regarding the root causes of the detected performance problem can be used to assist the redesign efforts. An effective solution can be devised and used to eliminate the identified performance anomaly. In the following sections performance antipatterns are introduced and our framework is described.

5.1. Performance Antipattern as Design Patterns

Design pattern is a formal way of documenting best practice approach in software development and system architecture design. The idea stems from the architecture field
in which experts are able to record reusable problem-solving designs. Ordinary users can easily apply relevant patterns in their application context to solve complex system design problems. The documented solutions are represented in a pattern language, which addresses a description of the solution to the problem, and the benefit gained from applying the process. In computer science, especially software engineering, a milestone was marked when a book on Design Patterns was published by Gamma et al. [44], promoting the reuse of software design elements in the context of object-oriented software design. Published design patterns are categorized in three groups: creational, structure, and behavior related patterns so that the user can easily locate the relevant ones to apply to the process. Since then, especially in the object-oriented engineering field, the idea of design pattern application has been widely adapted in the software design and development process.

As more applications of design pattern occur, more publications of design patterns in the system and software engineering field appeared. From languages such as Java, C/C++, C#, Ruby, and Ajax, to real-time embedded architectures and systems using UML, and to the broader applications in security, parallel, and distributed enterprise computing, design patterns are quickly accumulating, aiming to assist practitioners in doing the best practice in system development and producing good result.

Since the usability of design patterns is still fairly abstract, they are not easily adapted and applied in practice. Computing environment or context can be thought of as a multidimensional attribute set which has great impact on the execution of applications and systems. To better match the problem description, design pattern has to provide specific context information in which the design pattern is intended. In most cases, the context provides detailed descriptions in natural language to identify the scenario where the pattern is applicable and not applicable. Designers can look up the patterns and see if the scenario matches. Once found, they can apply the solution as best practice to assure the result of the design is in fact best possible. Although patterns are promising and of great help in system development, some gaps between practice and application still exist. One of the obstacles results from the activity of identifying the exact context and matching the scenario to the system. The
Another example of insufficiencies of original design patterns is the issue of performance. Software design patterns can be thorough in treating functional design problems, but they do not address other aspect of the design. This leads to another gap in design pattern application. Although the solutions to the design problems optimize the components of the system while building, they do not give clues to the quality of the design. In other words, only functional enhancements are ensured, whereas non-functional properties such as availability and reliability are not fully covered.

The noted design patterns in early publications addressing the performance issues in software systems is by Smith et al [74]. Principles of performance-oriented design are used as strategies in the development life cycle. They are embedded in the guidance of fundamental design practice which is later documented as performance patterns. Although performance patterns proposed in system performance engineering are to address the performance issues, they are presented at higher levels, while the context can only be determined after the implementation of the pattern has been chosen.

Instead of following the same format of design patterns, the performance patterns are published with the other side of the aspect, which documents the potential bad practices that lead to poor performance. They tell us what not to do and how to fix a problem when it appears. Such patterns are called antipatterns. Performance antipatterns are similar to design patterns in that they document recurring problems, but state the scenarios from the opposite side of best practices. If the scenarios match with a performance antipattern, the predictive outcome of performance can be poor. The solutions of how to avoid the pitfalls are documented as solution descriptions analogous to best practices in design pattern. The advantage of adapting performance antipatterns over performance patterns in practice is that they are easier to apply and are clearly guided due to the more concise coverage of the
scenario description space. However, antipatterns inherit similar obstacles that are common with design patterns in general. It is still not as straightforward to apply and gain the benefits from the solution. In this work, the focus is on targeting performance antipattern in the software development, and proposing approaches and tools to make the pattern application process more performance aware.

The main source of performance antipatterns are the works by Smith and Williams [72] [73] [75]. These antipatterns are presented to be adapted in a more general context than the others which may involve specific domain such as embedded real-time systems. Most antipattern related literature refer to these patterns as examples used in their research. Similar to the format of design patterns, documentation of antipattern consists of the name of the pattern, the problem it addresses, and the best solution to solve the problem.

The first step in applying antipatterns in the design process is to extract the problem description and discuss their feasibility for detection in real systems. Since the research community frequently refers to these as fundamental antipatterns, there are additional attributes that highlight their usage features. For example, to be able to detect the existence of performance antipatterns, values of performance indicators have to be acquired to decide whether a specific symptom exists. Some of these can be determined by just a single value, while others require multiple samples over time. The former can be categorized as Single Value (SV), and the latter as Multiple Value (MV) antipatterns. In this work, these annotated attributes of a performance antipattern are summarized as detectable features (DF). Detectable feature is the extraction derived from the problem description statements, which serve as the essential indicators to check for the existence of the pattern.

Not all of the detectable feature extraction is feasible without the involvement of specific design decisions from the context domain. There are antipatterns without clear-cut indicators as to whether the problem matches. Most of the cases in this category need extra interpretation of design concepts and decisions made by developers. One exemplified exception is the god class (or Blob) antipattern. The god class antipattern is an example of a badly designed class, which eventually does all the work and handles excessive messages. To
determine the appropriateness of the design, and the criteria of the performance threshold for the application, designers have to define the value of thresholds according to the domain feature of the system, and the experiences with the application. Although judging the existence of performance anomaly involves understanding of the way object classes are involved in execution, the problem can be quantified by counting the number of information flows through the class. With the quantified indicator, engineers can decide a threshold which fits the application context to raise an appropriate detection call. With appropriate tools proposed in this work, god class performance antipatterns are detectable.

Another example that involves design decisions is excessive dynamic allocation antipattern. It occurs when the application unnecessarily creates and destroys objects during its execution. The more than frequent number of object creation can cause severe performance overhead when allocating the memory space for the object. The behavior can be quantified by modeling the memory access behavior. The quantified indicator for memory usage over time can tell whether or not the symptom exists. Although memory modeling is not the focus of our work, our tool can easily be extended to measure and discover memory behavior so as to facilitate the detection of the symptoms [42].

To pave the way for applying solutions to solve performance antipatterns, the problem description is interpreted to extract the forces seen in the pattern description. Associated forces are defined as influential factors (IF) that play essential roles in affecting the outcome of applying performance antipatterns. Associated factors extracted from each antipattern will be used as the clues to the root causes if they are detected. It is also noted that the forces can be extended as new forces discovered from new archive. In a general system and software development context, the factors include Design (D), Algorithm (A), Configuration (C), and Threading (T).

**Design:** Design factor is concerned about software objects and how well they are established in the design and implementation. It often relates to the policy in the design of resource sharing and recycling, as well as the arrangements of steps and paths of computing. Different ways of the layout in the design results in different
computing behaviors, and therefore performance outcomes.

**Algorithm:** Algorithm factor is distinct from Design in the way that same software components can apply different strategies to achieve the computation goal. The designer can adapt a strategic approach for computing and use different structures to manage data. Different complexities of the algorithms lead to different execution times.

**Configuration:** Software development usually leaves options for configurations to let the user fine tune the behavior of the application to fit the usage expectation. While systems are ready to run, different management policies with corresponding configuration options can lead to different performance behaviors.

**Threading:** Multitasking has been one of the frequent models used in software systems to cope with the complexity in parallel and distributed environments. Thread, as an abstract execution unit, plays key role in carrying out the tasks along with other peer threads. Individual thread behavior, thread coordination, and management policy play significant role in affecting the overall system performance.

Problem style and solution description appearing in performance antipatterns is less than formal. More often, concise and plain natural language is used to address the documented scenario. Although easy to grasp for the reader, it is far from easy to directly embed them for an automatic detection and reasoning. To make performance antipattern more useful, root cause diagnosis should be included in the detection and solution process. Root cause identification is locating the problematic spot in the system which suffered from the symptoms described in the pattern description. The problematic spot has to involve system resource consumptions but fails to keep up with the required performance pace. It can be inferred that the forces documented in the pattern are the causes that influence the operation on the resource. Involved resource (IR) is defined to represent the hot spot of a performance antipattern. Involved resources are derived from interpreting the scenario compiled from problem and solution descriptions of each pattern. The identification mapping is straightforward and without ambiguity.
<table>
<thead>
<tr>
<th>Antipattern Name</th>
<th>Problem Description</th>
<th>Solution Description</th>
<th>DF</th>
<th>IF</th>
<th>IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unbalanced Processing</td>
<td>1) Occurs when processing cannot make use of available processors. 2) the slowest filter in a pipe and filter architecture causes the system to have unacceptable throughput. 3) when extensive processing in general impedes overall response time.</td>
<td>1) Restructure software or change scheduling algorithms to enable concurrent execution. 2) Break large filters into more stages and combine very small ones to reduce overhead. 3) Move extensive processing so that it doesn’t impede high traffic or more important work.</td>
<td>1) UT (SV) 2) TH (SV) 3) RT (SV)</td>
<td>1) D, C 2) D 3) C</td>
<td>1) CPU 2) CPU 3) CPU</td>
</tr>
<tr>
<td>The Ramp</td>
<td>Occurs when processing time increases as the system is used</td>
<td>Select algorithms or data structures based on maximum size or use algorithms that adapt to the size</td>
<td>RT (MV)</td>
<td>D</td>
<td>CPU</td>
</tr>
<tr>
<td>More is Less</td>
<td>Occurs when a system spends more time trash ing than accomplishing real work because there are too many processes relative to available resources</td>
<td>Quantity the thresholds where trash ing occurs and determine if the architecture can meet its performance goals while staying below the thresholds</td>
<td>TH (MV)</td>
<td>T, C</td>
<td>CPU, MEM</td>
</tr>
<tr>
<td>God Class</td>
<td>Occurs when a single class either 1) performs all of the work of an application 2) holds all of the applications’ data Ether manifestation results in excessive message traffic that can degrade performance</td>
<td>Refactor the design to distribute intelligence unfirmaly over the application’s top-level classes, and to keep related data and behavior together</td>
<td>TR (SV)</td>
<td>D</td>
<td>CPU, NET</td>
</tr>
<tr>
<td>Excessive Dynamic Allo-</td>
<td>Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.</td>
<td>1) Recycling objects rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.</td>
<td>MA (SV)</td>
<td>D</td>
<td>MEM</td>
</tr>
<tr>
<td>cation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-Lane Bridge</td>
<td>Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently. Other processes are delayed while they wait for the turn.</td>
<td>To alleviate the congestion, use the Shared Reoueces Principle to minimize conflicts.</td>
<td>QL (SV)</td>
<td>D, C</td>
<td>CPU, NET</td>
</tr>
<tr>
<td>Traffic Jam</td>
<td>Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.</td>
<td>Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.</td>
<td>RT (MV)</td>
<td>C, T</td>
<td>CPU</td>
</tr>
</tbody>
</table>

59
Table 5.2. Performance AntiPatterns with Undetectable Features

<table>
<thead>
<tr>
<th>Antipattern Name</th>
<th>Problem Description</th>
<th>Solution Description</th>
<th>DF</th>
<th>IF</th>
<th>IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unnecessary Processing</td>
<td>Occurs when processing is not needed or not needed at that time</td>
<td>Delete the extra processing steps, reorder steps to direct unnecessary steps earlier, or restructure to delegate those steps to a background tasks</td>
<td>N/A</td>
<td>D</td>
<td>N/A</td>
</tr>
<tr>
<td>Circuitous Treasure Hunt</td>
<td>Occurs when an object must look in several places to find the information that is needs. If a large amount of processing is required for each look, performance will suffer.</td>
<td>Refactor the design to provide alternative access paths that do not require multiple looks, or to reduce the cost of each look.</td>
<td>N/A</td>
<td>D</td>
<td>N/A</td>
</tr>
<tr>
<td>Falling Dominoes</td>
<td>Occurs when one failure causes performance failures in other components</td>
<td>Make sure that broken pieces are isolated until they are repaired</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Empty Semi Trucks</td>
<td>Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.</td>
<td>The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity Design pattern provide more efficient interfaces.</td>
<td>N/A</td>
<td>D</td>
<td>NET</td>
</tr>
<tr>
<td>Tower of Babel</td>
<td>Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML.</td>
<td>The Fast Path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using the Coupling performance pattern to match the data format to the usage patterns.</td>
<td>N/A</td>
<td>D</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The default resource used in the framework includes: CPU, MEM, NET, and DISK. Users should be able to add additional resources as more performance antipatterns are included. Also, users can further refine resources to lower levels of the system architecture as needed. For example, multiple processors can be identified uniquely to observe the workload precisely. In describing communication protocols, the identification of network interfaces and their specific ways of sending and receiving data flows through network ports can be referenced as needed to fully address the pattern of the behavior.

Table 5.1 presents the detectable performance antipattern list. These detectable ones
form the elementary performance antipattern in the framework. Each performance antipattern is summarized with its name, problem description, solution description, detectable features, influential factors, and involved resource as discussed above. Abbreviation used in the table are listed as follows: Utilization (UT), Throughput (TH), and Response Time (RT). The antipatterns that have no detectable features are listed in Table 5.2 for reference. The undetectable ones can not be included in the framework, because they are largely influenced by design factors, which are dependent on the design. And, as there is no obvious detectable feature, in most cases, there is no involved resource item that can be identified either.

5.2. Design Processes with Performance Antipattern

Detection of performance antipattern is a form of performance debugging. The purpose of the performance debugging is to make the system perform in a reasonably responsive manner and meets the requirements. In many scenarios, the symptoms of performance failure include unsatisfiable response time, which may be caused by bottlenecks due to scarce resources or workload congestion. The goal of debugging is to locate the problematic spot and to eliminate the root cause of the anomaly. In most system and software life span, the debugging activities are continuous activities along with iterative software development life cycle. Analogous to general debugging activities, performance debugging activity should also be embedded in the development process and run parallel with the development processes to ensure the performance is on the right track. Taken from a generic modeling process, life cycle phases are put in order from requirement analysis and design to implementation and deployment testing. The life cycle is always iterative to make incremental improvements for each round. The goal of the debugging process is to locate the bug in earliest possible stage, so that the problem can be resolved sooner to avoid expensive overhaul cost, which would be the case if discovered at a later stage. During the development process, engineers extract the required information to create models that assist in analyzing the design and planning for further verification and testing. These models are related to functionality of the system, and are used in validation and verification debugging purposes. Performance debugging, on the other hand, requires further information to facilitate performance analysis and evalua-
In software performance engineering [71], model-based performance analysis approach is adapted to generate performance data. They are created with the information about the architecture of the system, the capacity of its components, and the expected behavior of the system. Additional estimations such as request types and potential workloads are also needed for the modeling. The derived performance models are then used to produce quantitative numbers such as time duration, system utilization, and throughput. These values are used to serve as indicators formally known as performance indices. The combination of these indices is used to predict the possible performance outcome once the system is running.

In performance modeling, software architecture models are used to represent the backbone of the system structure. The preliminary performance measurement of the system can be obtained by examining the system architecture. Architecture models can be created based on the topology and attributes of the layout. The model can be transformed into its corresponding performance model by carrying relevant information about attributes such as capacities of components and their communication flow. Preliminary performance models can be solved by using queueing networks. The output of the analysis is a set of performance indices associated with the corresponding preliminary design. It serves as a performance indicator for engineers allowing them to gain insight of the predicted attributes of the system. Engineers can check the estimated performance outcome against the requirements of the system, and make decisions as to whether it meets the desired criteria or not. Engineers can use the information from the prediction, to either redesign or enhance the system to optimize the performance satisfaction.

The performance modeling process is depicted in Figure 5.1. The process is formally split into three phases. The modeling phase is the main stage of the system and software life span. Regular software models are built by following the orders of life cycle phases. This software modeling phase in practice is overlapped with performance modeling, because the updated performance attributes are gathered from the modeling activity as soon as the latest design revision is available. The second phase is analysis, and its goal is to create corresponding models for performance analysis and prediction. In this phase, model-to-
model transformation is taking place. System and software models are transformed into performance models with information such as designated architecture, its topological layout, and available resources. In the analysis process, performance indices are obtained by solving the performance models using queueing network tools. These indices are used as indicators to forecast performance scenarios. Performance indices are interpreted in the third phase called refactoring. The goal of refactoring is to reflect the latest performance attributes and determine the satisfaction of the design in terms of performance qualification. If the performance does not meet the requirement, feedback can be generated to initiate design changes according the interpreted results to resolve the performance issues. Engineers are obliged to check the predictive performance indices, and respond accordingly with changes to ensure the performance acceptance. Analogous to software life cycles, the performance modeling process should proceed iteratively in an incremental order to synchronize with the original software model, create and analyze performance models to generate up-to-date performance indices, and give feedback with design changes for a better performance.

In the performance engineering process, the goal is to detect performance anomaly in the design and resolve the issue effectively and precisely. However, performance indices
in summary can only provide the location of the problematic components and signal the anomaly. To be able to come up with a change of plan, the practitioner must look into the design knowledge of the system to find the cause and estimate the performance penalties accordingly. It is difficult for performance experts to reason using only performance indices if the system being built is relatively new. This is where performance antipattern comes in to fill the gap between interpretation and feedback generation, especially in the refactoring phase. Running parallel with interpretation steps, antipattern detection engine can be installed to assist performance antipattern identification. Once detected, the known solutions can be provided as feedback suggestions to remodel the system.

Antipattern detection mechanism largely depends on the problem description to discover instances of performance anomaly in the system, while feedback adjustments depend on the description of solution description. Ideally, the performance antipattern mechanism should be easy to adapt and build the engine for detection and solution when one follows the documentation of performance patterns. Once an instance is detected, performance antipattern comes with the solution description which can advise or blend with other feedbacks from different sources to generate the feedback. Figure 5.2 depicts the integrated process of software development modeling and performance modeling process. Both of the processes are synchronized in the modeling phase due to design and implementation revision. In the refactoring phase, mechanisms of antipattern detection is integrated to assist in identifying documented performance problems and generate solutions accordingly as feedback for redesign. The integrated process is synchronized incrementally and iteratively with the software modeling.

Although the promises of performance antipatterns, or design patterns in general, are great, the intricate nature of documenting a scenario and its environment in a computing system makes direct application of antipatterns difficult. If the goal is to put it into automatic practice, there are many gaps and challenges. One example of deficiency is recognition of the context where the performance antipattern exists. For instance, in unbalanced processing-extensive processing, the problem description states “the extensive processing impedes overall
response time.” It leaves to the discretion of engineers to realize what exactly the response time change is and how it should be modeled in the specific application. Another example in the ramp, the statement like “processing time increases” in the problem description, has to be determined by the engineers as to what is the significance of time indices in modeling so as to detect the symptom in the specific application.

Another noticeable hurdle in applying performance antipatterns is getting the solutions as feedback. Inherited from generic design patterns, solution description is one of the essential parts in pattern documentation that carries the key expert knowledge. The secret to completing the performance modeling process largely depends on the precision of solutions, which enables the debugging process to tweak the design to be cleared from performance pitfalls mentioned in the antipattern. The problem with context ambiguity, similar to the counterparts of performance antipatterns in detection mechanism mentioned above, appears in the solution description as well. For example, in more is less, to determine whether the architecture can meet its performance goals by staying below the thresholds, one has to decide the appropriate value for threshold to use as part of the solution. The threshold value is not only affected by the underlining architecture attributes; it also can be affected by the
degree of discretion which in turn depends on the context of the application. Another example, in Traffic Jam, one of the solutions is to provide sufficient processing power to handle the worst-case load. The processing power adjustment is an open issue to be determined to remedy the bottleneck. These examples show that applying performance antipatterns in the refactoring phase would need reasoning tools to assist the feedback generation in practical problems. In other words, once an instance of antipattern has been identified, applying the solution description to generate feedback for system improvement is not straightforward. Tools that can reason about the context for the specific system should be available to enable the reasoning process of refactoring. This is where profiling approaches are proposed and appropriate tools are implemented, which are described in the next sections.

In summary, performance modeling process is one of the essential aspects in the process of the system and software development. It should go hand-in-hand with development life cycle to ensure the quality of the performance. Interpretation of the indices from solving performance models requires experienced experts to discover and suggest solutions. In practice, system engineering and development involves various stakeholders such as designers, analysts, programmers, and administrators. They are all participants in the development life cycles, but may not happen to be performance experts in most cases. Therefore, it is promising that the performance antipattern mechanism provides assistance to spot problems and bugs with the help of accumulated knowledge documented by experts. However, the problem of context ambiguity as we identified above hinders the direct usage to the application. This is known as refactoring dilemma; there exists complexity to identify, manage, and discover the system and software context to match the performance antipattern description, which prevents antipatterns from being directly used.

To solve the refactoring dilemma, an approach is proposed to assimilate performance evaluation in the software developing process into performance antipattern detection. Before stepping into the performance modeling process, preliminary performance evaluation should take place. The preliminary evaluation provides not only initial performance information but also prepares the context that the system is built in. The first performance insight would
give practitioners clues about current performance status of the system, while the established context clears the ambiguous description by providing practical data for reasoning. In this work, it is also proposed that the preliminary evaluation be comparable to the antipattern detection mechanism. Especially, in the refactoring phase, the feedback generation should be able to provide useful solutions to spot root causes and remedy performance anomaly. The feedback should be easily applied in the software development activities. Furthermore, both the preliminary evaluation and feedback generation report should be able to provide fine-grained information that engineers can use for reasoning purposes in their own design and analysis approaches. In the next section, an approach and its tools are presented to assist the performance modeling process in solving the refactoring dilemma.

5.3. Performance Baseline

Before going to the specifics to the activities of performance antipattern detection, we first observe performance debugging in general. To detect if bugs appear, one has to judge if the criteria of the performance outcome is acceptable. The judgment can be made by comparing the observed performance to a predefined standard which usually represents the system requirements. If the indicator of observed performance is higher then the requirement, the conclusion of the debugging will not show any problems. It can be inferred that the design of the system with designated capacity will meet the performance specifications. On the contrary, if the performance indicator is lower then the requirements, the system is said to have performance bugs. In this case, the system has to be refactored to see if changes can be made in order to improve performance and meet the required specification. Without loss of generality, assuming we focus on the limited resource as the root cause, the performance dissatisfaction should be resolved by increasing capacities. However, in practice, cost is a key consideration in design activity. There are always tradeoff considerations between performance enhancements and cost. We argue that if resource allocation is bounded or has reached limit, the performance dissatisfaction may not be resolved without significant reengineering. In some cases, it is better to build a new system by laying out new architecture plans.
To assist in verifying the basic performance indices against system requirement and current version of the system, preliminary performance evaluation should be readily available. We introduce the concept of composite performance metric which serves as the indicator of preliminary performance metric named Performance Baseline (PB). The purpose of PB is to find the average performance of the system and its components, and give engineers a heads up about context information of the system in practice. We define PB as a set of performance metrics of software components or their integration and cooperation, such as an application or subsystem. Applications or systems run on the computing environment such as platforms with embedded resources. These resources include processors, memory, disks, and communication interfaces, where integration of system architecture and software components together provides computing services. The capacity of the platform depends greatly on the quantity and class of its constituencies and how they interact. To understand and reason about the relationship between performance and capacity one will need to make a great effort in observing the system behavior and the resource usage. The complexity is even higher as systems tend to increase their capacity by applying methods of replication and scalability. In practice, within any development stage, the complexity could discourage engineers to pay extra scrutiny to oversee the performance development. This is where PB comes in to assist the preliminary performance evaluation. In later sections, the approach will be addressed and tools are implemented to enable the practical usage of PB. The performance indices produced from PB gives the performance summary of target objects engineers choose to observe. Performance baseline serves as the performance indicators which represent reasonable behavior of the target software entity under the observed context. A preliminary diagnosis of performance anomaly can be identified by checking the observed performance against the baseline.

The abstraction of PB and its relationship to system requirements is depicted in Figure 5.3. The x-axis represents component over time, and y-axis represents the performance index. The dashed line represents the performance baseline observed from the current status of the system. There are two requirement specifications A and B that represent relatively
Figure 5.3. Overview of Performance Baseline and its Relation to Performance AntiPattern

low and high performance demands. For component C1 under A, the observed performance indicator is above the required point P. The performance is classified as satisfactory. For component C1 under B, the observed performance indicator is under the required point Q. The performance is classified as unsatisfactory, and therefore identified as a pending performance bug indicating that some correction efforts are needed.

There is also a dotted curve to conceptually represent applying one performance antipattern solution outcome. For component C1 under B where antipattern is detected, the application of the solution can enhance the performance to point R, where requirement B at point Q can be resolved. In practice, there is also the scenario that the requirement bar is high and it is unlikely that an engineer can apply antipattern solutions in order to resolve the issue. For component C2 under requirement B, the observed performance baseline is ranked at the lowest performance in this context. It is unlikely to meet the requirement and therefore serves as an indication of performance dissatisfaction. Furthermore in the context, we observed that the application of performance antipattern solution may not totally overcome the deficiency, since the requirement bar may be set to high. It is essential that engineers are well informed about the limitations of the reengineering activity so that the
Since performance result comes from the integration and cooperation of all entities from software and system architecture, an abstract structure is proposed to help us identify essential elements of the performance forces and express how they organize and play the roles in the computing. A system and all its entities is modeled in a structure called System Abstraction Layers (SAL). The system modeling layers consists of three divisions from top to bottom: software, platform, and hardware. Table 5.3 depicts the contents of each layer and their design abstraction. Each design abstraction represents the forces from a system developing activity that play a key role, affecting the behavior of the element in the layer.

In the software layer, systems are realized by the integration of sub-systems, each of which is responsible for a specific purpose in terms of functionality module. For each subsystem at the software layer, entities of software in terms of components and libraries are composed to create the sub-system that carries out the execution of the computing services. Each component in this setting is executing the tasks designated for it. The abstraction of task control can be related to configuration if the tuning mechanism is available from the software entity. The real execution in the software layer to carry out the tasks of a component is given to the elementary execution entity known as thread. The abstraction is

<table>
<thead>
<tr>
<th>Software</th>
<th>Layer</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sub-System</td>
<td>Integration</td>
</tr>
<tr>
<td></td>
<td>Component</td>
<td>Composition</td>
</tr>
<tr>
<td></td>
<td>Task</td>
<td>Configuration</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>Execution</td>
</tr>
<tr>
<td>Platform</td>
<td>Layer</td>
<td>Abstraction</td>
</tr>
<tr>
<td></td>
<td>Middleware</td>
<td>Resource Management</td>
</tr>
<tr>
<td></td>
<td>Operating System</td>
<td>Scheduling Policy</td>
</tr>
<tr>
<td>Hardware</td>
<td>Layer</td>
<td>Abstraction</td>
</tr>
<tr>
<td></td>
<td>Execution Unit</td>
<td>Processing Element</td>
</tr>
</tbody>
</table>

requirements can be properly met.
also compatible with the implementation using only the process unit in which each process is treated as a special main thread.

Many software systems need to take advantage of using services from middleware to ease the complexity of developing and deploying applications. Middleware is used to manage the communication resources and hide the interaction details from the users, especially for distributed systems. In a broader sense, it also manages the server resources by regulating how control and information flow is distributed under the planned architecture topology. Under the middleware, it is the operating system that provides services for resource management and process scheduling for different tasks in a server. Platform layer is about resource management where the system and its software entities reside and access to the computing resources.

The bottom layer of the server under the platform layer is related to hardware component organization. It is where the practical performance is measured. Performance revealed from the hardware layer depends on the grade of components installed to facilitate running the system. Utilization of hardware components can be acquired from this layer which includes processors, memory, network, and Disks among others.

With the defined conceptual layers in SAL, with each of them relating to a specific design abstraction, we can describe a performance scenario flexibly both at higher and lower levels of the layer. A high level scenario expression can be refined and mapped to its corresponding lower level counterparts. Through the process of mapping, we can identify the related elements in each layer and reason about the forces associated with them. This lays out the foundation for the case of performance anomaly, when the root cause elements can be identified in the hierarchical approach. To accommodate the context information, the structure of layers can be formulated into syntax as description language. The performance context of a system or application can be expressed.

The framework also consider the performance metrics that correspond to the features of each SAL layers. In the software layer, the number of components can be recorded with the mechanism and communication protocol they use to interact with each other. For all
entities in the software layer, different configurations contribute to different context settings, which in turn affects the performance. Data about configurations related to performance can be recorded. If configuration involves a number of thread pools, or the settings on thread behavior policy, it will also affect the performance of multitasking. The context of a computing scenario can also be identified with the recorded setting. Overall, time duration between events is the most used performance metrics, and can be applied in any layer of SAL. We argue that the unit of the PB metrics can range from lower level counts such as number of instructions or basic blocks, to higher level representations such as number of software components or hosts that provide services. With the fine granularity and flexibility to handle performance metrics, it enables engineers to explore feasible design possibilities. Engineers can reason about the performance scenarios, and explore solution consequences in more detail. Within each phase of the development life cycle, engineers can flexibly choose the measurement granularity of performance metrics that best benefits the software development.

In summary, performance baseline contributes to the activity of performance debugging in the software development process as follows:

- In software modeling phase, PB collects preliminary performance indices to produce the first evaluation directly related to the design and implementation of the system. During deployment and maintenance phase, PB can be the auxiliary source of validation and verification.
- In the performance analysis phase, data gathered by PB can contribute as input to modeling as needed.
- In the refactoring phase, PB can be integrated to detect and solving performance antipattern. During the feedback, the reasoning information can be seamlessly translated to relate them to PB.

In the following sections, a combinatorial profiling mechanism is proposed to establish the performance baseline. The profiling mechanism will enable practitioners to easily adapt PB to detect and resolve performance antipatterns with context management. The associated
tools and their implementations will be discussed.

5.4. Suggestive Profiling

The purpose of profiling in our framework is to identify the performance anomaly and to locate the root causes. Once performance antipatterns appear in the performance modeling, the practitioner should be able to detect and get the suggestive solutions depending on the current context of the system in order to remedy the problem. To gain the causality reasoning capability, the proposed profiling mechanism is set to conform to the system abstract layers. The profiling mechanism can also serve as the backbone of the performance baseline because they are all compatible with SAL structure. For the assistance role the profiling mechanism plays in the performance debugging process, it should be compatible to both software development and the performance modeling process, making it easy to adapt in any phases in the process. The profiling mechanism should be easy to setup for performance testing and evaluation. Since it has been made compatible with performance baseline, it also has to assist in performance antipattern detection and provide feedback, which is the main purpose of the process. With these design requirements in mind, the following discussion provides the design rationale and discuss the components consisting of the profiling mechanism, the context where they can be applied, and how they can be operated to serve as baseline.

Conventional software profilers usually focus on the source code or its corresponding binary execution to get statistical measurement of the software package or library. The information includes frequency and duration of routines such as function calls. The goal of conventional profiler is program optimization. System profilers focus on the observation of resource usages of the server and the software that is installed on top of it. They monitor the state of hardware resources such as processors, and report consumption summaries. All the profiling mechanisms above are essential to our purpose. However, more precision and reasoning structures are needed to achieve our goal. We need

- Specific timeline information that can identify not only spatial hot spots but also the temporal features.
Table 5.4. Suggestive Profiling Method in the System Abstract Layer Context

<table>
<thead>
<tr>
<th>System Context</th>
<th>Profiling Method</th>
<th>Suggestive Profile Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem, component</td>
<td>Path-Oriented</td>
<td>Alternative Path options</td>
</tr>
<tr>
<td>Thread</td>
<td>Thread Behavior</td>
<td>Thread Behavior comparisons</td>
</tr>
<tr>
<td>Middleware</td>
<td>Networking Profiling</td>
<td>Request traces and communication protocol verification</td>
</tr>
<tr>
<td>Hardware</td>
<td>System Resource Profiling</td>
<td>Physical and Abstract resource summarization</td>
</tr>
</tbody>
</table>

- Profiling information from one aspect that can relate to another, such that a reasonable mapping can be inferred.
- Profiling information that can be summarized and compared with associated context.
- Communication and distributed settings that can be integrated in the profiling mechanism.

To this end, we put together the profiling mechanism needed to fulfill the requirements of our purpose. In particular, our goal is to assist in performance antipattern detection, as well as feedback generation. In addition, software systems more than often run in networking environments; the profiling mechanism needs to flexibly accommodate and adapt to distributed settings as well. In Table 5.4, the profiling mechanisms are categorized into contexts that are matched to system abstract layers. Each of the profiling method is given a brief description followed by its suggestive profile method. The purpose of the method is to explore other available options in the same context level of the system to give leeway in enhancing the performance result or avoiding bad practices. The practitioner can take advantage of the suggestive profiling approach to explore design options for better result. In the performance refactoring phase for antipattern detection, solution direction can be hint by exploring the suggestive solutions methods.

In the following subsections, each one of the profiling mechanism used in the framework will be discussed and the relationships among them in the overall suggestive profiling will be described.
5.4.1. Path Profiling

The framework adapted the terminology of Path Profiling from data flow analysis [10], and the pathwise decomposition concept from path-oriented analysis [38]. The concept of path-oriented profiling is based on the measurement that execution of different program paths can result at different time slots. If we can make the most common path execute fast, the response time may be shorter. Path profiling also provides insights on improving performance by revising the chances of executing certain paths, or improving the efficiency of the path. If the frequency of execution of a path is relatively high, the savings in execution time can become significant. By using the technique, inefficient path of the program can be identified. The dependency of paths and associated components can be revealed for further analysis of the paths.

Path profiling can be seen mostly at the level of software components and libraries, and in the programs. In the software layer, execution path is the lowest unit of refinement for the software system. For each thread, path profiling is also essential for discovering performance problems. It serves as the backbone for analyzing elements from the software aspect.

The concept of execution path can be extended to accommodate information flows. Information flow tracking in the program is to understand the pattern of execution paths as certain requests are being executed. It can also be extended to include communication routes that connect the execution path between server nodes. The high level view of path profiling can be observed at the subsystem level where interactions between clients and servers use different routes. This view can help practitioners to understand the entire scenario of the execution. Alternative routes between them may be the result of dispatching policy or adapting flexible algorithms to react traffic congestions.

5.4.2. Thread Behavior Profiling

Thread is a sequence of instructions and the representation of a logical computational unit, which can be scheduled to run by the operating system. Multi-threading is a widely adapted programming model in software systems, whose creation overhead is lighter than the
creation of a full process. A pool of working threads can be initialized before the real workload picks up and be ready to respond without delay. In system architecture, adapting the thread model also gains the benefit of executing true concurrency in a multicore environment. Thread Behavior profiling is about the observation of thread creation, running, destroying, and management of all threads. At system architecture level, processor affinity can be monitored as multithread programming specifies the arrangements. The combination of resource distribution and the management policy such as number of threads and their running priorities affect the overall performance. Threads can also be viewed as another form of dynamic path, because every thread runs on its own copy of instructions. The observation and summary of individual threads can be performance indicators of how well they coordinate and cooperate.

The context of thread profiling is at task level where the system adapts multithreaded programming model to carry out designated services. Depending on the features of the application, threading system usually provides the tuning mechanism as a configuration option so that administrators or users can adjust the behavior among threads to improve their performance. Thread is the lowest logical task unit that we can monitor in the profiling. It provides the flexibility of measurement in both higher and lower levels of the system. At the higher level, an end-to-end performance scenario can be profiled by integrating thread behavior profiling in each subsystem with the information flows. At the lower level, each task and its resource usage by a specific thread can be analyzed. The flexibility of thread monitoring facilitates the whole system profiling at a fine-grained level.

5.4.3. Network Profiling

Networked system has become the infrastructure for every computing system no matter where it resides, either in enterprise clusters, virtual hosts, or Clouds. The complexity of interaction patterns between servers increases exponentially. As the network becomes the computing platform, it is inevitable that one needs to learn profiling or monitoring in order to understand the performance impact of the overall system. Our model classified the networking in the context of middleware to accommodate network and includes proxy, router,
programming middleware, and other network topologies such as multi-tier and clustering. Networking profiling focuses on getting information about request calls and responses, and the underlining communication protocols. Measurement information can be about the number of requests at higher level, and the number of network packets at lower level. It can also look into the data that packets carry, and profile the characteristics of the request message. Performance of the network activities can contribute to either the network interface capacity such as queueing buffer, or the handling speed of the server.

Network profiling information can be referenced by its connected systems and software components to make the design decision more informed. It can also contribute to the construction of analytical models such as queueing networks with more realistic estimation of queue lengths and arrival rates. Finally, for information flow analysis, the networking profiling facilitates visualization of the activities in detail, and provides the overall picture of the networked components and systems.

5.4.4. Resource Profiling

Conventional resource profiling is about profiling physical resource usage. The resource here refers to the hardware of system architecture in the server. The higher capacity the server has, the higher the performance. In practice, cost, overhead, and limitations can affect the choices and ability to obtain more capacity. Resource profiling gives information about resource characterization of the server and its computation capacity. It usually hints at the potential performance outcome of the system.

One puzzle of resource profiling is in the context of software abstraction layer, where engineers often want to know precisely how much resource has been consumed by a specific software entity. For example, if the load a working thread contributed to a processor load can be calculated, then the total resource requirements can be estimated given the number of threads. Physical resource monitoring alone cannot fulfill the job because it is hard to get a clear-cut workload of a working thread without being affected by the influences from other programs and the management operations in the platform.

To estimate the resource consumption of a software entity, instead of applying mea-
measurement and estimation from system utility, we turn to resource consumption estimation using number of instructions executed. Resource profiling in software entity can be abstracted to the lowest level of computation using instructions before putting them into operation. These instructions are the ones that are consuming resources. Therefore, the system resources that need to accommodate these executions can be estimated. Depending on the type of resource, a processor’s workload can be approximated by the number of instructions running on it. The instructions can be further categorized by the type of memory access in which they consume different instruction cycles. For a storage disk, number of I/O accesses resulting from the execution can also be profiled into resource consumption. Advanced resource modeling such as register and cache behavior can be devised to extract the measurements. We name this type of resource profiling as abstract resource profiling in contrast to the physical resource profiling method discussed above. The mapping between software entity and the abstract resource can be clearly identified and the resource consumption can be reasonably estimated.

The application of resource profiling can be applied to most of the system. Software systems usually adapt monitoring mechanism to extract the information from system hardware components. Abstract resource profiling will need the other profiling mechanisms in the suggestive profiling to supply the measurement. All the conventional resources in the antipattern domain, such as CPU, Memory, Network, and Disk, can participate in both resource profiling.

5.4.5. Request and Workload Profiling

Workload from clients of any system and software plays an influential role in the outcome of the performance. Although the profiling method is not categorized into any system abstract layer, it is a vertical effect that every part of the system will experience consequential impact. Depending on request patterns, one subsystem or component may have a larger workload then that of the others due to their computation role in the system. If the workload exceeds planned capacity of the components, a bottleneck would occur. Similar considerations apply to the performance impact of threads, in that they may spend longer
time processing requests, and the throughput may suffer. In the context of middleware, interaction between servers may take longer under heavy workload may have wait for response. In system architecture layer, the computing workload coming from the above has a direct result on resources and overheads when switching to meet the services.

The request and workload profiling provides information about the impact in each context of the system. Engineers can evaluate the scenario of request pattern in each focused context independently. The profile relating only to an individual context can be used as specific source of information to focus on that particular design improvement. On the other hand, the profiler can characterize the workload, so that the design of the system can be adjusted flexibly if possible to increase the performance. For example, if the profile of the request is cpu bound, engineers may consider distributing them as evenly as possible to available servers. Another alternative solution occurs when cpu bound requests prefer to be sent to a high speed server with better equipped cpu capacity. We also note that both physical and abstract resource profiling information can be used to characterize workloads. This information can help engineers understand the impact of requests.

In practice, request types and their temporal patterns are dynamic, and the workload characteristic is not known a priori. The suggestive profiling mechanism can be used at deployment to selectively monitor requests, and create the workload profile associated with the context. The profiling can select time periods or focus on a specific component for performance monitoring.

5.5. Detection and Solution Suggestion Process

To be able to use suggestive profiling in performance antipattern detection and solution feedback, it has to understand performance baseline. Performance baseline is the summary of current performance metrics of the system. It can be used for performance debugging to check against requirements. Preliminary performance evaluation can be obtained by establishing the baseline of the target system or components. The content structure of the baseline is compatible with the system abstraction layer, in which path-oriented, threading, networking, and resource profile is recorded. In each context of the profiling, performance
metrics such as time duration and process utilization are available for verification. Performance baseline can be created for every element in each context of suggestive profiling method including subsystem, component, thread, network, and hardware component. Depending on the needs of debugging activity, engineers can zoom in on targeted components and their interactions when high level information is not enough. In short, performance baseline is an agile performance filtering and debugging tool used in the software development process to collect targeted performance snapshots.

For distributed and networked systems, two compositional operations which provide integration capability to performance baseline of networked servers are proposed. \( \odot \) represents PB concatenations of two servers communicating directly. For example, assuming S1 and S2 are the servers with interactions, then the whole performance baseline can be created by connecting the two together as \( S_{1pb} \odot S_{2pb} \). Another type of server connection is branching, when there are multiple servers to choose from. \( \oplus \) represents performance baseline concatenations to two or more servers. For example, assuming S0 is the server that has to optionally choose to communicate with S1 or S2, the composition performance baseline can be represented as \( S_{0pb} \odot (S_{1pb} \oplus S_{2pb}) \). Other combinations of server topology can be represented by the composition using the two basic operations.

With performance baseline as the debugging framework used in the system development process, activities in performance modeling processes can share the data it collects. Since both of the processes are synchronized, the performance metrics collected are reflected in the latest status of the system. In the refactoring phase, performance antipattern detection and solution suggestion feedback mechanism can be executed with the help of suggestive profiling. In detection, performance baseline is accessed to extract the metrics of the detectable features needed with each antipattern. The value of detectable features can be checked to see if the symptom appears. If not detected then the antipattern does not exist, and no action is needed. If a matched is detected, the solution may be applied to solve the performance anomaly in the system. Antipattern solution can point to the problem spots and give approaches to resolve the problem. As discussed in previous sections, refactoring
dilemma exists because we need more clues so as to generate detailed feedback for redesign to eliminate the anomaly. To close the gap, suggestive profiling can be used to narrow down the root cause, and provide refined causality.

We recall that the suggestive profiling method consists of computing path, thread behavior, networking, and resources in the layered context defined in SAL, and each profile can be evaluated independently. In order to close in at the root cause, we can examine the suspicious context provided by profiling. Within the root cause, performance metrics gathered by the profiling mechanism are verified against the performance antipattern symptoms to discover corresponding solutions. For example, if the root cause specifies a component is the bottleneck in the system, we can further analyze time duration of its computing path, and make a specific solution suggestion in refactoring. Although we reason at the specific component level, levels higher and lower than the root cause context can also be inferred to seek the potential redesign options. For example, if the root cause is about thread performance, execution paths at a higher level or resources at a lower level can be inferred as the relevant factors. Solution suggestions can therefore provide more relevant information in details.

5.6. Implementation

The suggestive profiling tool using Pin tool are implemented, and an associated data analytic framework for performance debugging, antipattern detection, and solution suggestions are created. Pin [53] is a dynamic binary instrumentation framework which provides APIs to enable application analysis and program instrumentation. This tool can trace each executed instruction of an application for path-oriented analysis. It also provides other instrumentation points including basic blocks, routines, images, and complete application. These abstractions can be used to identify call graphs, accesses of libraries, and inter and intra component communication, which can easily fit in the system models.

Pin also provides process and threading primitive APIs which facilitates individual thread profiling. In the event of thread creation, destroying, or changing state, Pin sends out the notification with a callback. For each thread, execution context such as registers and
its data can be traced. In the thread profiler, we also include the ability to detect forking of processes, and tracing correspondingly both parent and child thread. The threading capability enables our tool to watch over each individual thread behavior in the application.

Network profiling of the suggestive profiling tool is implemented by monitoring system calls. Pin tool provides system call APIs which capture the number of system calls and their arguments made to the operating system. Our tool then focuses on the communication relevant calls such as socket, read, write, open, and close, and the communication ports they use. This allows us to collect the pattern and messages as the data of network profile.

For abstract resource profiling, Pin provides an excellent fit to trace the execution at instruction level. It is straightforward to profile the abstract resources with the counts of executed instructions or basic blocks. Instructions can also be analyzed by types of memory access which further refine the resource profiling capability to estimate the costs of executing the application. For physical resources, Pin provides the API interface to emulate the usage of registers and memory. Programmers can plug in the model for their specific profiling approach. However, there is no direct access to CPU and Disks among other system components. There are two approaches we used to add this information. Our tool can either acquire CPU information by adding instrumentation for query, or the information can be directly accessed from the operating system. Because most servers monitor system resources, the profiling data can be easily available.

While suggestive profiling runs online with the application, all the profiling data collected from the execution is recorded and can be analyzed in order to participate in different stages of performance debugging. For regular performance debugging, performance baseline as summary function is implemented. It gathers all the profiling data, and gives a summary of performance indices such as average response time in each context. For performance antipattern detection, the current indices of detectable features from performance baseline are extracted to check with the symptom. A comparison function is implemented in the tool for antipattern detection. In the case of performance antipattern discovery, further root cause analysis should be carried out to find a solution. A difference function is implemented to
compare current suggestive profiles with the antipattern thresholds at each context level, and provide a summary for output.

In the framework, our tool includes utilities in the following to facilitate suggestive profiling usage:

- Data collection, processing, and management are separated by different profiling methods.
- Communication between software components and systems is profiled with the help of protocol plugins. Each plugin specifies the pattern of interactions.
- System resource monitoring facilitates logging and analysis.
6.1. RUBiS Benchmark

RUBiS is an auction site prototype modeled after eBay.com. This benchmark is used to evaluate application design patterns and performance scalability of servers. It is a three-tier production system to emulate interactive sessions of clients who use the web site. Types of user activities can be distinguished by three roles: visitor, buyer, and seller. The core functionalities implemented for client emulations of the auction site are browsing, bidding, and selling. There are 26 interactions that can be performed from client’s web browser.

The workload in RUBiS is categorized by the weight of reading and writing. The browsing mix is made up of read-only interactions, while the bidding mix includes 15% read-write interactions. A defined workload is prepared by a probability transition table to guide the random selection of web pages a user is clicking next from current page activity. Workload generation emulates the behavior of a number of users, each one of them represented by a independent user session. Every user session follows the setup of the workload mix to emulate a sequence of requests launched by the user.

RUBiS provides an open source client emulation program, platform setup, and monitoring management. It also uses popular open-source software as components of the system. The setup lets users freely modify the application to fit their particular needs for running the benchmark. Users can take advantage of the setting mechanism to allow them to plug in the proposed framework and tools for experiments. The architecture overview of RUBiS is depicted in Figure 6.1. The server tiers include a web server front-end to accept requests from remote users. The requests are then passed to the application server that handles the

![Figure 6.1. RUBiS Benchmark Tiers](image-url)
business logic, and retrieves the requested data from the database back-end as needed. RUBiS can be setup for different design patterns using various server components. Mechanisms such as PHP, Servlet, and EJB are the ones commonly used.

6.1.1. Experimental Setup in Production System

RUBiS was setup on Xen 3.1.2 virtual machines hosted on Dell Optiplex 960 with 4 CPU and 4GB RAM. Each virtual machine runs on the allocations of one virtual CPU and 512MB RAM. Each virtual server is connected to a virtual network interface with a unique network address. The virtual network connection is created by an ethernet bridge, and a DHCP server is setup to assign unique network address to each virtual server.

RUBiS was installed with Apache2 httpd, JBoss AS 4.3.2, and MySQL as the web server, application server, and database respectively. Sources are available for each of these server software packages, and allow us to build and install from the compilation of the source codes. The suggestive profiling Pin tool was installed on the web server to generate the performance baseline for the web server, and to demonstrate performance antipattern detection and solution feedback suggestions. The suggestive profiling tool can be run on any of the servers, and on the targeted application. On each server virtual machine, we measure the load using sysstat utility to collect CPU, memory, network, and disk usage every one second. All the traces and logs generated from the suggestive profiling Pin tool, and the system utility measurement logs are collected afterward to avoid interferences with the workload of the server.

To run an experiment, the configurations of virtual servers and their applications are automatically setup with the help of our utility tools. They are installed separately from virtual servers to assist in data collection, processing, and analyzing the execution result. After the results are collected, performance summarization, comparison, and differentiation can be carried out by using the utility tools. In the following case study, the EJB option was adapted as the default setup in running RUBiS.
### Table 6.1. RUBiS Request Types with Identification Numbers

<table>
<thead>
<tr>
<th>Number</th>
<th>Request Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Home</td>
</tr>
<tr>
<td>2</td>
<td>Register</td>
</tr>
<tr>
<td>3</td>
<td>RegisterUser</td>
</tr>
<tr>
<td>4</td>
<td>Browse</td>
</tr>
<tr>
<td>5</td>
<td>BrowseCategories</td>
</tr>
<tr>
<td>6</td>
<td>SearchItemsInCategory</td>
</tr>
<tr>
<td>7</td>
<td>BrowseRegions</td>
</tr>
<tr>
<td>8</td>
<td>BrowseCategoriesInRegion</td>
</tr>
<tr>
<td>9</td>
<td>SearchItemsInRegion</td>
</tr>
<tr>
<td>10</td>
<td>ViewItem</td>
</tr>
<tr>
<td>11</td>
<td>ViewUserInfo</td>
</tr>
<tr>
<td>12</td>
<td>ViewBidHistory</td>
</tr>
<tr>
<td>13</td>
<td>BuyNowAuth</td>
</tr>
<tr>
<td>14</td>
<td>BuyNow</td>
</tr>
<tr>
<td>15</td>
<td>StoreBuyBow</td>
</tr>
<tr>
<td>16</td>
<td>PutBidAuth</td>
</tr>
<tr>
<td>17</td>
<td>PutBid</td>
</tr>
<tr>
<td>18</td>
<td>StoreBid</td>
</tr>
<tr>
<td>19</td>
<td>PutCommentAuth</td>
</tr>
<tr>
<td>20</td>
<td>PutComment</td>
</tr>
<tr>
<td>21</td>
<td>StoreComment</td>
</tr>
<tr>
<td>22</td>
<td>Sell</td>
</tr>
<tr>
<td>23</td>
<td>SelectCategoryToSellItem</td>
</tr>
<tr>
<td>24</td>
<td>SellItemForm</td>
</tr>
<tr>
<td>25</td>
<td>RegisterItem</td>
</tr>
<tr>
<td>26</td>
<td>AboutMe (auth form)</td>
</tr>
<tr>
<td>27</td>
<td>AboutMe</td>
</tr>
</tbody>
</table>

#### 6.2. Request Type and Workload Profile

One of the orthogonal products that can be easily generated from suggestive profiling is the data on request types and workload patterns. In performance modeling, workload is one of the essential factors that affects the performance of the system, and should be taken into consideration in the software developing process. The workload information is naturally captured by the instrumentation of every sent and received request packets. The communication event can be noted as the starting and ending of the service time, as the request comes and leaves the system. When targeting a specific application or component, one can zoom in on the analysis for the corresponding request.
As mentioned in previous section, 26 request types are defined and categorized into browsing and transaction attributes in RUBiS. When running the benchmark, our framework can specifically recognize each and every client request packets by giving it a unique request identification. As a request comes to any profiled software object in the system, its identification can be annotated for further analysis. A brief list of the request types and their corresponding numbers used in the experiments is presented in Table 6.1. Figures presented in the following will use the list of numbers to represent its request types. RUBiS was tested using different workloads with the number of users ranging from 100 to 1000. Figure 6.2 depicts the run using default browse-only request set, while Figure 6.3 depicts the run using default specified in transition table.

Figure 6.2. RUBiS Benchmark Using browse_only_15 Workload
6.3. Performance Antipattern Detection and Solution Suggestion

6.3.1. Unbalanced Processing

As a request is received by the Apache Web Server (WS), the service is forwarded to the JBoss Application Server (AS) for data retrieving and processing. To flexibly distribute the workload and increase the processing capacity, server replication is used. WS can dispatch the request to any one of the AS according to its load balancing policy. In Apache WS, the Apache Tomcat Connector called mod_jk is used to enable the redirection. For each replication server, it is named as a worker server node in mod_jk. For each worker, specific settings for communication between WS and AS can be established to use designated network port, communication types, and load balancing factor among others. These configurations together with capacity of system architecture and its components will influence the performance outcome of the system. When the outcome of performance comparison does
Figure 6.4. One Web Server Dispatches Requests to Two Application Servers in RUBiS Noted with Performance Metrics

not live up to the expectation, the problematic spot has to be investigated and dealt with in order to discover the root cause. In this case, it can be a great challenge for searching multidimensional configuration combinations.

Without loss of generality, we discuss the case consisting of two AS replications that a WS can invoke. The setting is depicted in Figure 6.4. P1 and P2 are the performance indices resulting from request dispatching from WS to AS1 and AS2 respectively. PB1 and PB2 have been noted in dashed line to represent documented performance baseline recorded in AS1 and AS2 respectively. There are three general cases:

(A) P1 > P2 : Performance P1 is better then P2. The metric differences between the two dispatches are significant overtime.

(B) P1 == P2 : Performance P1 and P2 is at equal. The metric differences between the two dispatches are rare or insignificant.

(C) P1 < P2 : This is the same analogy as in the first case.

In case (A), Unbalanced Processing antipattern can be detected, due to the symptom of the unbalanced processor utilization or divergent response times observed between AS1 and AS2. At this point, we have different considerations about what to pursue next:

(1) Both P1 and P2 are satisfied. There is no performance violation in this case, so performance debugging can be skipped. Performance antipattern can be applied optionally to improve the unbalanced server setting.
(2) P1 is satisfied but P2 is not. P2 can be compared with PB2 to see if AS2 has enough system capacity to meet the performance requirement. If the capacity is not available, performance debugging will have to focus on increasing the capacity of the processing power in order to accommodate average workload demands. On the other hand, the system is not using best scheduling that led to the uneven dispatched processing. The system solution to the antipattern can be applied to discover the suggestive solutions for recovery.

(3) P1 and P2 are not satisfied. P1 and P2 both can be compared with PB1 and PB2 respectively, to see if AS1 and AS2 are qualified candidates to meet the performance requirement. Individually, if AS1 or AS2 falls short compared to the baseline, it is subject to increased capacity before any performance antipattern detection mechanism can be applied. Even if only performance baseline PB1 is not satisfied by P1, chances are P2 cannot be applied to antipattern detection due to the solution includes the involvement of AS1. The performance has also suffered if it is ambiguous as to whether or not it was the reengineering efforts on the influential factors that lifted performance to the required capacity.

In case (B), Unbalanced Processing antipattern cannot be detected due to the lack of detectable features. In this case, PB1 and PB2 can be used to guide the preliminary performance evaluation of the AS1 and AS2. If they pass the performance baseline, and meet the expectation, there will be no action taken until next round of performance debugging. If both of them fall short, they are handled as case A3 mentioned above.

In the cases of A1 and A2, we apply problem description to detect antipattern by using detectable features. Once located, the corresponding solution can be applied by checking influential factors and the involved resources. According to the solution either design or configuration is the influential solution factor in the antipattern. With the help of suggestive profiling, we compare the number of requests dispatched to AS1 and AS2. In the case that the distribution is not even, the most likely root cause is configuration that causes the imbalance in the processing. Once we are focused on the configuration of request dispatching,
we locate the load balancing settings. Specifically, the lbfactor in the Apache WS workers configuration is responsible for the load balancing policy.

On this occasion, the appropriate set of value to resolve the unbalanced scenario is not known yet. We turn to suggestive profiling for further information. At this point, two scenarios can be encountered:

- Topology of PB1 and PB2 exists. Apply the difference function to the current profile against the existing performance baseline, and discover the suggestive solutions. For example, the ratio of lbfactor variable setting between PB1 and PB2 can be proportioned to be the suggestive adjustment for the current setting.

- Topology of PB1 and PB2 does not exist. The new performance baseline of PB1 and PB2 in this scenario can be established. The ratio of the new topology can be obtained by comparing the configuration between PB1 and PB2 versus the ratio of P1 and P2. For example, the lbfactor of P2 can be adjusted towards the setting of P1 in proportion to the ratio of PB2 versus PB1.

Experiment using default transition workload with 500 users whose requests served by two JBoss ASs and one Apache WS is presented with different ratios of lbfactors. Figure 6.5 depicts the number of requests dispatched from WS to either worker1 of AS1 or worker2 of AS2. The load balance ratios are marked as the ticks on x-axis. We observed that the number of requests to worker1 and worker2 are approximately proportionate to the weight of the lbfactor. Figure 6.6 depicts the time duration of requests dispatched from WS to worker1 or worker2 with different dispatching ratios. The representation is not liner, but it reflects the scenario in which the preferred node spends more time processing because of the biased setting. Both the dispatching number and the time duration with different ratio are interrelated.

6.3.2. More is Less

In this case study, the meaning of more is less is extended to the context of deciding the appropriate configuration of threading policy. System and software are often evolving
Figure 6.5. Dispatched Number of Requests to worker1 and worker2 of JBoss

Figure 6.6. Average Execution Time of Requests Dispatched to worker1 and worker2 of JBoss
due to revisions, upgrades, and reorganization. As a scenario emerges, maintaining the performance level at greater or equal to its previous ones can be essential to its availability and reliability. Users can follow the suggestions provided by the manufacture. In practice, every server is situated in its own unique environment so that not all situations can be covered specifically in the manuals. An analogous scenario can be applied to the problem of context ambiguity in the performance antipattern description. It is inevitable that a general solution to performance antipattern cannot solve the problem on the spot. A practitioner may not be familiar with all the options for the settings, therefore it is necessary to create a tool that assists with searching for a solution.

Apache WS supports pluggable concurrency models called Multi-Processing Modules (MPM). There are worker, event, and prefork options that users can choose to use at runtime. Worker MPM uses multiple child processes with many threads each, while prefork MPM uses multiple child processes with one thread each. Event MPM is an alternative to workers with minor alternative strategy in request processing. Prefork is a special case to worker MPM. In general practice, the worker module is the one often used, and thus it is chosen to be adapted in the experiment. In the worker module, configuration variables to control the size of the thread pool and the number of concurrent working threads are defined in http-mpm.conf file of Apache, and are briefly listed here:

- StartServers: initial number of server processes to start
- MinSpareThreads: minimum number of worker threads which are kept spare
- MaxSpareThreads: maximum number of worker threads which are kept spare
- ThreadsPerChild: constant number of worker threads in each server process
- MaxRequestWorkers: maximum number of worker threads
- MaxConnectionsPerChild: maximum number of connections a server process serves

In this experiment, the RUBiS benchmark with different sets of configurations was run, and the outcome of the performance baselines and their differences were observed. Figure 6.7 depicts a test run with 800 users using various sets of worker configuration shown in the legend. The numbers in the legend correspond to the order of StartServers, MinS-
Figure 6.7. RUBiS Benchmark with 800 Users Using Various Sets of MPM Configuration

pareThreads, MaxSpareThreads, ThreadsPerChild, MaxRequestWorkers, and MaxConnectionPerChild. For each request types from the benchmark, the corresponding average response time is shown.

6.3.3. God Class

Modeling of abstract resources can be very useful when a precise estimation of physical resources is hard to achieve. In god class the detectable feature is a threshold of a programming class. The threshold suggests that if the amount of jobs a class executed is over a reasonable limit, the performance will likely be degraded. The influential factor that can alter the scenario is design. The threshold has to be determined with the appropriateness related to the application attributes. It poses a great challenge to a designer, as well as a
performance expert, because it is difficult to figure it out just by looking at the program. It is especially hard without real workload to test. Moreover, even if we can count the usage frequency of a class in a program, it is still impractical to estimate the amount of resources needed to support the class.

With the help of suggestive profiling tool, we can observe the information flow of the program. More specific investigation is possible by zooming in on targeted class or function and refining the entity to a basic block or an instruction level. At these lower levels, abstract resource profiling can take place to estimate program execution costs with the frequencies provided by the information flow. Together with the testing workload, the total resource in abstraction needed by the targeted entity can be realized.

In the following, the abstract resource profiling of the Apache httpd server used in the RUBiS is demonstrated. Developer documentation of httpd states that, all requests pass through `ap_process_request_internal()` in request.c of the web server. We want to observe the information flow and its frequencies when a real workload is used. Before checking the flow of information to the targeted function from suggestive profiling tool, symbol table of the `ap_process_request_internal()` function is extracted from the ELF (Executable and Linkable Format) of httpd to find the static address of the function. This information is used to acquire the structure of instructions in execution order. Request flow information collected from suggestive profiling tool is checked with the static structure of the function to produce the request flow graph in practice. We also note that the capability of following the analysis approach is restricted to open-source software.

Figure 6.8 shows a snapshot of the disassembly from `ap_process_request_internal()`. Basic block units in the function are identified to record the execution paths. Request flows in a working thread from an exemplified benchmark run are analyzed against the static structure and presented in the basic block unit as depicted in Figure 6.9. The number of times an execution traverses between basic blocks is annotated. Engineers can review the design with the performance data, and make changes accordingly.
Disassembly of section .text:

```
08081ad4 <ap_process_request_internal>:
  08081ad4:  55      push   %ebp
  08081ad5:  89 e5    mov    %esp,%ebp
  08081ad7:  83 ec 48  sub    $0x48,%esp
  08081add:  8b 45 08  mov    0x8(%ebp),%eax
  08081ae0:  8b 45 08  mov    0x8(%ebp),%eax
  08081ae4:  8b 45 08  mov    0x8(%ebp),%eax
  08081af1:  b8 01 00 00 00  mov    $0x1,%eax
  08081af6:  eb 05      jmp  0801ae0  
```

Figure 6.8. Disassemble Snapshot of ap_process_request_internal() Function in httpd
Figure 6.9. Snapshot of Request Flow Graph of ap_process_request_internal() Function in the httpd Web Server
CHAPTER 7

CONCLUSION

This dissertation consists of two topics of software engineering research in performance engineering. The topic of the first half is Web service computation, and the topic of the second half is performance antipattern detection and solution suggestion. The focus of the two parts is about performance evaluation of software systems. However, the two parts in research are independent.

In the first half of the work, a framework for composing web-services using both functional and QoS properties is proposed. To accommodate the service quality annotations, WSDL descriptions of web-services was extended so that non-functional or quality of service parameters can be associated with the service. As a proof of concept, APIs for locating web-services based on both functional and non-functional properties are developed. Ontologies that can be used to select and compose web-services have been developed. For the purpose of composing non-functional properties based on component services, new reasoning engines must be developed. Different non-functional properties may require different reasoning engines. This work also demonstrated how performance properties can be composed using queuing engines. As argued in this research, most services can be composed either in series or in parallel, and can use a queuing engine to derive the performance properties of the composed service.

To facilitate the framework with other Web service composition capabilities and standards, the QoS-aware framework is extended again using BPEL. With the foundation of WSDL extension to annotate non-functional properties, Web services can be selected based on both functional and non-functional (QoS) properties whether or not they are presented using BPEL. This dissertation described the process for publishing and discovering services which meet requirements in the standard service oriented architecture. As shown in the reasoning statements, services in BPEL description can be seamlessly accommodated in the framework. By adapting BPEL and BPEL4Chor for service composition, we can discuss the
feasibility of adapting service orchestration and choreography in the framework. It turns out that the proposed framework was able to accommodate these Web services composite frameworks naturally.

This research also devised transformation rules for converting BPEL into appropriate queuing networks which can be used by the layered queuing network tool and can compute performance metrics. To illustrate the applicability of our framework in deriving QoS properties of composed services, performance properties such as throughput, response times, and utilization are used. A building security case study was used to demonstrate this process. Although the focus is on performance evaluation in this work, the framework can also be used to compute other QoS properties such as reliability, security, and availability, with appropriate rules for converting BPEL logic into corresponding models and tools for obtaining QoS properties from these models.

In the second half of the research, approaches for performance antipattern detection and solution were proposed. Software performance antipatterns are similar to design patterns, but address problems in terms of what to avoid and provide solutions on how to fix them. This thesis first analyzed each software performance antipatterns to understand the context described. During the analysis, detectable features, influential factors, and involved resources are extracted for each of the antipatterns to lay the foundation for their detection and root cause diagnosis.

Before the antipattern detection step in the performance modeling process, a performance metric called performance baseline was proposed. The purpose of the baseline is to gain insight into preliminary performance evaluation, to see if the system is on the right track, and whether or not it has the potential for improvement.

This research also discussed the context ambiguity problem, which states that there are still gaps between detection and application of solution because performance antipatterns depend on context. It is impossible to know how precise antipatterns match and the performance result will be after applying the pattern solution. To close this gap, it is necessary to develop a tool that can assist in the detection and solution suggestion process.
To establish the framework for building the tool, a system layered abstraction model is proposed to assist with pinpointing elements in the system. The model enables us to identify components and entities in the system and their mappings to lower or higher levels of the system. The mechanism enables the framework to flexibly determine what root causes exist.

In this dissertation, the suggestive profiling method is proposed. It is based on the system layered abstract model, and consists of path-oriented profiling, thread behavior profiling, networking profiling, and system resource profiling. For each profiling method, a suggestive profile method is discussed to give possible alternatives in re-engineering for better performance. Request and workload profiles can also be generated through the suggestive profiling tool. This technique is used in the solution suggestion during refactoring phase of performance engineering, and is synchronized with software development cycles. The process for applying this approach in performance antipattern detection and solution feedback is proposed in this work.

The suggestive profiling tool and the framework utility tools have been implemented using Pin tool. The dissertation presented case studies of performance antipattern detection and solution suggestion for three different antipatterns: unbalanced Processing, more is less, and god class. RUBiS benchmark is used in the execution of these experiments.

There are limitations in matching some performance antipatterns with detectable features, and thus they cannot be detected directly. Most of these undetectable antipatterns are due to design decisions. Thus an intimate knowledge of the designs can help in the detection and elimination of those performance antipatterns. If the design decisions can be systematically codified, then it will be possible to extend our framework to other performance antipatterns.

In the future, we plan to further analyze factors influencing antipatterns in different domains including high-performance computing, e-commerce or workflow data management, and extend the framework with appropriate tools. We also plan to make the framework cloud-ready, so that general performance antipatterns in the computation of distributed
systems can be categorized, detected, and resolved systematically.
REFERENCES


[10] Thomas Ball and James R Larus, Efficient path profiling, Proceedings of the 29th annual


[19] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and FRANCESCO Lo Presti,


[28] Francisco Curbera, Frank Leymann, Tony Storey, Donald Ferguson, and Sanjiva Weer-


[49] Noura Limam and Raouf Boutaba, *Assessing software service quality and trustworthi-


[59] Arun Mukhija, Andrew Dingwall-Smith, and David S Rosenblum, Qos-aware service


[70] C. Schröpfer, M. Schönherr, P. Offermann, and M. Ahrens, A flexible approach to service


