RECOMMENDING SERVICE REPAIRS

by

Joshua Church
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_______________ Dr. Amihai Motro, Dissertation Director

_______________ Dr. Daniel Menascé, Committee Member

_______________ Dr. Alexander Brodsky, Committee Member

_______________ Dr. Huzefa Rangwala, Committee Member

_______________ Dr. Stephen Nash, Senior Associate Dean

_______________ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _______________ Fall Semester 2014
George Mason University
Fairfax, VA

Recommending Service Repairs

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Joshua Church
Master of Science
George Mason University, 2007
Bachelor of Science
Bradley University, 2003

Director: Dr. Amihai Motro, Professor
Department of Department of Computer Science

Fall Semester 2014
George Mason University
Fairfax, VA

Copyright © 2014 by Joshua Church
All Rights Reserved

Dedication

This is dedicated to Emily.

Acknowledgments

I would like to thank everyone who expressed encouragement and support while I worked on my dissertation. In addition, I would like to thank my committee for their constructive comments and suggestions. I would like to give a special thank you to my advisor, Dr. Motro for his time, advice, and guidance throughout this entire process. I much appreciate it. Finally, I would like to thank Emily for her patience.

Table of Contents

Page

List of Tables . . . viii

List of Figures . . . ix

Abstract . . . x

1 Introduction . . . 1

1.1 An Open Problem and Research Focus . . . 1

1.2 Service Repair Problem . . . 2

1.3 A Summary of Research Contributions . . . 3

1.3.1 A Model for Information Services . . . 3

1.3.2 A Definition for Service Similarity . . . 4

1.3.3 A Heuristic for Sampling Small Behavior Tables . . . 4

1.3.4 A Heuristic for Improving Substitution Performance . . . 6

1.3.5 A Method for Recommending Substitutions in Service Compositions 7

1.4 Research Goals and Dissertation Outline . . . 8

2 Background . . . 10

2.1 Service Oriented Architecture . . . 10

2.2 Adaptive Software . . . 11

2.3 Code Search . . . 13

2.4 Service Substitutions . . . 15

3 A Model for Information Services . . . 18

3.1 Domain Expressions . . . 18

3.2 Services . . . 19

3.2.1 Behavior . . . 20

3.3 Pragmatic Adjustments . . . 21

3.3.1 Reducing the Dimensionality of Behavior Tables . . . 21

3.3.2 Exceptions in Behavior Tables . . . 22

4 A Language for Service Compositions . . . 24

4.1 Compose and Combine Operators . . . 24

4.1.1 Service Expressions . . . 27

4.2 Behavior of Service Expressions 28
5 A Definition for Service Similarity 32
5.1 Service Similarity 32
5.1.1 Similarity of Domain Elements 32
5.1.2 Similarity of Behavior 35
5.1.3 Service Exceptions 36
5.2 Directed Similarity 38
5.3 Similarity of Services with Different Outputs 39
5.3.1 Domain Alignment 43
6 A Heuristic for Small Sample Behavior Tables 45
6.1 Approach 45
6.1.1 Partitioning Services by Their I/O Domains 47
6.1.2 Eliciting Service Behavior Tables 47
6.1.3 Comparing the Services 48
6.1.4 Determining Minimal Test Cardinality 49
6.2 Evaluation 51
6.2.1 Classifying Weather Services 51
6.2.2 Classifying Stock Quote Services 53
7 A Heuristic for Improving Substitution Performance 57
7.1 Approach 57
7.1.1 Exhaustive Search 57
7.1.2 Metric Space Embedding 58
7.2 Evaluation 60
7.2.1 How Good are our Recommendations? 62
7.2.2 How Much Faster is the Target Space? 63
7.2.3 How Many Pivot Services are Needed? 64
7.2.4 How do Exceptions Affect Recommendation Precision? 65
8 A Method for Recommending Substitutions in Service Composition 66
8.1 Approach 66
8.1.1 Analyzing Repair Options 67
8.1.2 Repair Procedure 68
8.1.3 Present Recommendations to Programmer 69
8.2 Evaluation 70
8.2.1 Are Compositions of Behavior Samples Good Estimations for Behavior of Service Compositions? 70

8.2.2 Are Repair Recommendations Accurate? 73
9 Conclusion 74
9.1 Summary of Contributions 75
9.1.1 Summary of Issues Explored 76
9.2 Future Directions 77
9.3 Final Remarks 79
Bibliography 81

List of Tables

Table | Page

1.1 Summary of Contributions 9

6.1 Service Behavior 48

6.2 Weather: Rand Convergence 52

6.3 Stock: Rand Convergence 55

List of Figures

Figure Page

4.1 The *compose* and *combine* operators. 28

6.1 Heuristic for small samples of behavior 46

6.2 Weather: accuracy distribution 54

6.3 Stock: accuracy distribution 55

7.1 Precision at 4 neighborhood sizes 62

7.2 Precision at different number of pivots 64

7.3 Precision at 4 percentages of service exception 65

Abstract

RECOMMENDING SERVICE REPAIRS
Joshua Church, PhD
George Mason University, 2014
Dissertation Director: Dr. Amihai Motro

An increasing amount of data is provided by information services: Programs constructed with web services that return output from underlying data sources given input supplied in HTTP requests. Instead of the traditional SOAP/WSDL standards, this restricted type of service is designed in the Representative State Transfer (REST) architectural style in which service functionality is memoryless, stateless, and free of side effects. Indeed, a service-oriented approach to software construction offers many benefits, but there is one looming disadvantage. After deployment, services within the program may fail. In response, programmers may attempt to recover lost functionality with another service. Unfortunately, the repair process is a manual task where programmers must locate the failure and then repair it with a substitute; that is, another service with similar functionality. While techniques are available for identifying where the program is broken, an open problem remains: How to identify good substitutions?

To address this problem, recent research explores methods to automate aspects of the repair process, and thus support programmers while they debug failures. As yet, most automated repair methods assume programmers have access to program source code and so they focus on internal software changes. However we consider services to be black-boxes; that is, implementation details (such as source code or documentation) are unavailable.

Consequently, programmers cannot modify the failed service or rely on source code for insight into its functionality. Hence we focus our repair methods on recommending service substitutes and not on modifying source code. In other words, given a failed service, and assuming services are registered in a repository, the problem is to recommend another service (or set of services) that would be a good substitute for the failure.

Without source code, our idea is to describe service functionality by its behavior; that is, the observable relationship between its input and output. The thesis of this dissertation is service behavior can be modeled, sampled economically, and compared efficiently. Then, given a service defined in our model, we can recommend substitute services (from a repository) that behave similarly. In this dissertation, we propose a model for representing information services, we define a notion of similarity between two services, and we specify a language for expressing compositions of services. Then, we present a heuristic for extracting small samples of service behavior, a heuristic for improving the performance of our recommendations, and a method for making recommendations in the context of service compositions.

Chapter 1: Introduction

An increasing amount of data is provided by *information services*: Programs constructed with web services that return output from underlying data sources given input supplied in HTTP requests. Instead of the traditional SOAP/WSDL standards, this restricted type of service is designed in the Representative State Transfer (REST) architectural style in which service functionality is memoryless, stateless, and free of side effects [1]. For example, given a movie title, a service may return a list of theater locations, or given a gene's name, a service may return a description of its function. With information services, programmers may build custom views of data in *compositions* that pass output through a series of connected services. Indeed, as we see in scientific computing [2–4] or e-commerce [5–9], we can envision a *repository* of information services that catalogs data available to programmers.

1.1 An Open Problem and Research Focus

Certainly, a *service-oriented* approach to software construction offers many benefits, but there is one looming disadvantage: After deployment, services within a program may fail [10]. Such disruptions occur when a service becomes unavailable or when its functionality unexpectedly evolves [11, 12]. For instance, in an empirical study of publicly available services, almost all monitored services experienced some inactive time over the span of two months [13]. In short, this instability can be summarized succinctly in this manner: *What worked yesterday might not work today.*

In response, programmers may attempt to recover lost functionality with another service. Unfortunately, the repair process is a manual task. Namely, programmers locate the failure and then replace it with a *substitute*; that is, another service with *similar* functionality. While techniques are available for fault localization (i.e., identify where the code is

broken) [14, 15], an open problem remains: *How to identify good substitutions*?

To address this gap, there is recent interest in supporting programmers while debugging failures by automating aspects of the repair process. As yet, most automated repair methods assume programmers have access to program source code and so focus on internal software changes [16, 17]. However we assume services are "black-boxes"; that is, implementation details such as source code or documentation are unavailable. Consequently, programmers cannot modify the failed service or rely on source code for insight into its functionality. Hence we focus our repair method on substitute recommendations and not on source code modifications.

Without source code, we need an alternative description of service functionality. Thus our idea is to describe service functionality by its *behavior*; that is, the observable relationship between its input and output. To facilitate this, we assume either services have instrumentation to capture runtime invocations [18], or we can glean runtime invocations from deployment logs or other histories of service executions [19]. In this manner, we can sample functionality instead of constructing it manually with external representations such as tags, ontologies, or natural language text, which are often incomplete, ambiguous, or simply inaccurate [20].

1.2 Service Repair Problem

We begin by formalizing the service repair problem: Given a failed service s_0, and assuming available services are registered in a repository $\mathcal{R}$, the problem is to recommend another service s^* (or set of services) from $\mathcal{R}$ such that s^* would be a "good" substitute for s_0.

The thesis of this dissertation is service behavior can be *modeled*, *sampled economically* and *compared efficiently*. Then, given a service defined in our model, we can recommend substitute services (from a repository) that behave similarly. Therefore, in this dissertation, we investigate the problem of repair in service-oriented software, and present a model for information services, a definition for measuring the similarity between two services, and a language for expressing compositions of services. After explaining our model, we present a

heuristic to extract a representative sample of service behavior while keeping sample size small. Then, we apply our model to the task of recommending substitutes and present a heuristic for improving the performance of our recommendations. Finally, we expand our goal to the task of recommending service repairs in compositions of services.

1.3 A Summary of Research Contributions

In the course of our investigation, we focus our analysis on three major research issues: service behavior, service substitution, and service composition. In what follows, we isolate five challenges and describe a contributions toward each.

1.3.1 A Model for Information Services

Our statement of the problem assumes a repository of information services where each service is described by its behavior, so before we make any recommendations, we must develop a formal model for information services. The model is to be economical and general, providing structures for addressing the essential issues in the composition, substitutions, and repair of existing services. The challenge is to create a model that allows for significant automation of these processes, i.e., given a service described in the model, be able to recommend substitutions.

As a result, in Chapter 3, we present a model that abstracts services as *functions* from input values to output values. An important feature of the model is the input and output of a service could be *arbitrarily complex*. For example, a service that returns the value of a stock portfolio — given a particular date and a set of stock symbols and their corresponding quantities — has input which is a pair comprising a single value (date) and an arbitrarily-sized set of pairs (stock symbol and quantity). Towards this end, our model defines the syntax of service input/output parameters as domain expressions.

Another feature of the model is the functionality of services need not be represented externally — by means of natural language descriptions, sets of keywords, ontologies, and so on. Instead, functionality is inferred from observable behavior. Thus the *extensions* of

these functions, which we call *behavior tables*, provide the semantics of these services.

As behavior tables could be very large (even infinite), we make two pragmatic adjustments to our model. First, we assume only *samples* of the behavior tables are available. To address this challenge, our model establishes a common set of inputs in which meaningful *behavior table samples* can be extracted. Second, our model handles the situation when services may not respond with valid output given sample input values. We call these invalid outputs *exceptions*.

1.3.2 A Definition for Service Similarity

Our problem asserts we can recommend "good" substitutes for a given service. As such, our model of information services requires a formal notion of *service similarity*. This similarity definition must handle the structure of the output (a single value, a set of values, or a sequence of values from different attributes), and it must gauge both *output* similarity (when both services produce an output) and *coverage* similarity (when one service produces an output and the other does not).

Towards this end, in Chapter 5, we define a notion of service *similarity* based on observable behavior. Similarity is measured with distance *metrics*. Each basic domain (such as real numbers or character strings) is associated with a simple metric, and these metrics are then combined to create complex metrics that could be associated with domain expressions. Given two services with identically structured inputs and outputs, their similarity is derived from the similarity of their outputs a common set of inputs. Additionally, the definition is extended to deal with service *exceptions*: instances in which services might not provide valid outputs as expected. In Chapter 7, we apply our model and notion of similarity to the task of recommending service substitutes.

1.3.3 A Heuristic for Sampling Small Behavior Tables

Our model assumes the ability to extract meaningful samples of service functionality. This could be stated as a concept learning problem where the goal is to find a set of observations

that can be used to distinguish services from each other.

We highlight here some of the challenges that our behavior sampling heuristic addresses. First, it must generate unbiased service tests automatically and handle different service exceptions (e.g., no response, error message) gracefully. A test value may be acceptable as input but the services may not respond with output, thus exceptions require special treatment. Second, the number of service calls must be of the smallest size that assures an accurate representation. Indeed, the sample size reflects the dimensionality of our behavior, so our challenge is to reduce the number of dimensions and keep only a minimum number of tests. Hence we want to comprehend the difference between services using the smallest number of tests. This is because testing entails costly network traffic, but also because excessive testing might cause servers to block future access.

Towards this end, in Chapter 6, we present a heuristic to extract small samples of service behavior by means of *testing*. The objective is to determine the "ideal" sample size that successfully exceeds a predefined threshold of behavior quality and another predefined threshold for change. To this end, we design a sample strategy that iteratively increases the number of tests until a level of accuracy *converges*; that is, additional tests do not yield significant changes in behavior quality. These samples are obtained by querying the services with random values selected from their input and observing their output.

Are small behavior table samples meaningful reflections of service functionality?

To evaluate our model, we must demonstrate that samples of behavior indeed reflect the functionality inherent among services. That is, that services that behave similarly group together. Consequently, we measure behavior quality by comparing the number of agreements between cluster assignments derived from sample behavior tables and actual cluster assignments of service functionality. The idea is as the number of service calls increases there will be a corresponding change in service clusters. Then, we can compare these clusters to the true clusters and so measure the accuracy of our behavior samples. Thus we can gauge the difference between successive sample sizes, so when clusters change less than

a predefined threshold, we can stop the sampling process and establish a common sample size.

Our results indicate reliable behavior can be learned from relatively small samples where 10 to 32 tests can reflect service functionality with cluster accuracy ranging from 83% to 95%. Additionally, statistical significance tests validate results do not happen by chance. Hence given this small size, we could assume that this sample behavior is stored alongside the service in a repository.

1.3.4 A Heuristic for Improving Substitution Performance

Our thesis asserts we can recommend a service that can replace the functionality offered by another. Hence we address the issue of *service substitution*: How to best replace one service with another? Recommending the service (or group of services) that are most similar to a given service can be stated as a k-nearest neighbors problem.

A naïve solution to the problem of recommending substitutions is to compare the observable behavior of the given service to each candidate service. This search for a substitute is exhaustive (i.e., 'brute-force') where the search space includes all possible substitutions for each service. While this approach serves to define our objective, it is obviously inefficient. Indeed as the service-oriented paradigm gains popularity, the number of available services increases substantially, making the searches for substitutions more complicated and costly, and raising the need for heuristics to improve recommendation performance [21]. Hence our primary research goal is to develop heuristics that will arrive at high quality recommendations in acceptable time.

To avoid long computations at the time when the substitutions are needed, similarity measurements should be prepared ahead of time. Since metrics could be complex and the number of services large, the cost could be high. To overcome this challenge, in Chapter 7, we describe a heuristic that *embeds* the given set of services into a vector space, where the complexity of measurement is reduced dramatically.

Are substitution recommendations accurate and efficient?

The quality of our substitutions requires experimental validation. Using synthetic services, our experiments show the precision of our recommendations to be between 88% and 97% for individual services. In addition, our experiments show that the metric space embedding heuristic reduces recommendation time by as much as 98%.

1.3.5 A Method for Recommending Substitutions in Service Compositions

At times, a single service may not provide the desired information, but assume it may be computed in a composition built from services in the repository. In Chapter 5 and 7, we described how this service can be substituted with an alternative chosen from a repository. The new service must be input-output compatible with the failed service, and it should exhibit the closest behavior to it (i.e., maximize a service similarity measure that we adopt). Still, in Chapter 8, we consider the more complex situation in which the failed service is contained in a composition.

This new situation presents several challenges: First, we must consider the option that, instead of substituting the failed service with another repository service, we might get better results by substituting an entire *sub-expression* in which the failed service is contained (possibly even the whole composition). The new composition could be more efficient in that it would achieve similar functionality with fewer services. This requires the repair procedure to consider all the sub-expressions in which a failed service is contained.

Furthermore, recommending a good substitution cannot be done "locally", where the failed service (or a sub-expression that contains it) is compared with the new alternative. Instead, we must compare the behavior of the original composition to the behavior of the new composition that results. Obviously, in these processes we must have access to the behavior of compositions. The challenge is, unlike repository services, the behavior of compositions is not readily available, and must be *inferred* from behavior samples of its component services.

To express compositions of repository services, in Chapter 4, we define a simple language to create service expressions with the help of two operations, called *compose* and *combine*. Then, we define a transformation that takes services expressions and maps them to behavior tables.

In addition, in Chapter 8, we present a method that considers potential repair options and recommends a *ranked list* of repairs to the programmer. Obviously, there could be situations where no recommended repair is deemed acceptable since a service "similar enough" to the failed service could not be found.

Are substitution recommendations in compositions accurate?

Two aspects of substitutions in a composition require experimental validation. First, we must demonstrate that the compositions of estimations of behavior is a good estimate for the behavior of the compositions. The second aspect is the *quality* of our repair recommendations in composite services. Our experiment showed that when the list of recommendations is of length 10, in 85% of tests a repair that with the same semantics as the failed composition is recommended.

1.4 Research Goals and Dissertation Outline

The following four goals outline our investigation of the service repair problem. Together, they scope our analysis of the problem as well as guide our solutions. Our first three goals are to develop a formal model for information services, to define a notion of service similarity, and to specify a language for service compositions. Once a service is expressed in our model, the fourth goal is *repair*; that is, recommend substitutes when a service fails.

The rest of this dissertation offers a formal treatment of the service repair problem. To this end, we provide a detailed description of our service model in Chapter 3. In Chapter 4, we present a language for service compositions. A definition of similarity between services is presented in Chapter 5. A method for sampling service behavior, and a heuristic for

determining behavior sample size is presented in Chapter 6. Then, a method for recommending substitutions in a single service context is present in Chapter 7, plus a heuristic for improving the performance of service comparisons. Then, in Chapter 8, we describe our approach to analyzing repair options in compositions of services. In Chapter 9, we conclude with potential directions for future work and final remarks. Next, we begin with a broad description of the research areas most related to this dissertation in Chapter 2. The following table outlines where each contribution can be found in this dissertation.

Table 1.1: Summary of Contributions

Chapter	Contribution
3	A Model for Information Services
5	A Definition for Service Similarity
6	A Heuristic for Sampling Small Behavior Tables
7	A Heuristic for Improving Substitution Performance
4	A Language for Service Compositions
8	A Method for Recommending Substitutions in Service Composition

Chapter 2: Background

This dissertation focuses on the recommendation of service substitutes after a component service fails. Hence, we introduce a model and various methods to maintain functionality of deployed service compositions. It is, therefore, helpful to consider prior research in three active software engineering topics: adaptive software, code search, and service substitutions. In each area, we focus our investigation on service-oriented software. Next, we discuss briefly background information necessary to place our research in the context of these larger areas.

2.1 Service Oriented Architecture

SOA is an approach to software design that is characterized by the concept of *services* and three related activities: *discovery* of services, remote *invocation* of services, and *composition* of services [22, 23].

In service-oriented software engineering, services function as a computational wrappers around resources (e.g., information, algorithm, or hardware) [23]. In addition, services enumerate and describe resource operations available to clients. Typically, these operations are small in scope, perform a single task, and avoid long execution times. Services are implemented in a native programming language and made accessible to clients via well-defined interfaces such as Web Service Description Language (WSDL) files or Representational State Transfer (REST) forms. This separation of implementation and description allows clients to use services without regard to source code.

Service discovery is the activity of finding services given some query for functionality. To facilitate this, SOA assumes service repositories can store and manage information pertinent to the location and the description of registered services. Generally, the registration process is manual and uses the Universal, Description, Discovery, and Integration language (UDDI)

standard. Discovery unburdens service consumers from knowing where on the network services reside and what functionality services offer.

To invoke a service, the client provides input, the service executes locally with this input, and then it returns the output to the client. This message exchange occurs over HTTP and may be defined in many standard message formats called bindings. For example, Simple Object Access Protocol (SOAP) messages are a typical means for exchanging messages and more recently REST-based is an increasingly popular binding alternative that accepts input by standard HTML forms and returns XML documents. Generally, these binding methods follow remote procedure call principles intended to abstract operation invocation from implementation. It is this abstraction that offers SOA practitioners platform independence and programming language neutrality [23].

Service composition is the final related activity wherein programmers compose services together. This composition appears to clients as a single service. Generally, programmers build compositions manually by selecting desired operations from a collection of pre-existing services. To aid programmers in this activity, popular standards such as Business Process Execution Language (BPEL) are available to visually plan and model compositions. This planning is an iterative activity: Find relevant services and then sequence them together. Service reuse offers SOA designers architectural flexibility and service interchangeability to their applications.

2.2 Adaptive Software

Correction after unexpected software modification is also a theme in adaptive software research [24]. This field explores techniques to adapt software in response to changes to user needs, to run-time environments, or to program properties. In the context of service-oriented software, the goal is to maintain service functionality by reconfiguring its architecture (rearrange sequences of services such that different service compositions perform the same functional requirements [25, 25, 26]), by adjusting its parameters (refine the properties of

service in a composition [24, 27]), or exchanging its components (replace a single service in compositions with an equivalent service [28]). We call the third type of adaptation a substitution and it is the emphasis of our review.

Self-adaptive systems emphasize the selection of alternative services in situations when the runtime environment affects the continued operation of software [27]. In particular, self-healing approaches initiate service substitution when stated Quality of Service (QoS) requirements are no longer met. The issue of *when* to initiate substitutions is addressed in [29]; in contradistinction, our work addresses the issue of *which* services should be selected as substitutions.

Given a multitude of services that perform similar functions, this raises the issue of choosing the "best" substitute. There are numerous approaches that define the concept of best, but generally, the objective is to select the *most relevant* services from among a set of choices. In [28, 30–33], differences in quality of service (QoS) parameters are considered when choosing between equivalent services. Quality is measured by [28] as the greatest similarity between the QoS of a candidate substitution and the QoS expected of a given service composition, and in [33] quality is measured by utility functions that are defined by domain experts and given run-time performance data. In contrast, we emphasize differences in observable behavior and not quality of service.

Additionally, the previous works assume information about service functionality is given by external sources whereas we learn service functionality from run-time traces. For this purpose, our approach is facilitated by instrumenting software as it runs and then collecting invocation traces in a repository [18]. Of course, this type of information can also be gleaned from deployment logs or other histories of program executions [19, 34].

A second essential issue we address is the formal representation of service compositions. Similar to our approach, the research in [35] uses input-output behavior to model program behavior; however, it does not consider intermediate program states such that occur in service compositions. In addition, the research of [36] uses input-output behavior to compose loop-free programs built from a repository, but in contrast, its formal representation of

semantics is captured in constraints derived from output values whereas our representation of semantics is captured in tables of output values.

Finally, the work in [37] proposes a technique to find sequences of services that act as a 'workaround' for a failed component service. Like our research [37] supports repair of service-oriented programs; however, their technique requires knowledge as to which services are substitutable. On the other hand, our work uses services similarity to find service substitutes.

2.3 Code Search

Given a query service, our task is to find similar services from a repository such that one may replace the other. Towards this end, many approaches to code search adapt information retrieval techniques (e.g., classification schemes, free-text indexing, and relational databases) to indexing software artifacts [38–41]. These works assume textual descriptions of functionality are given. However services are generally offered without source code or documentation, and available WSDL files that describe service functionality are often incomplete, inaccurate, or ambiguous. For this reason, we use outputs from a system-supplied sample of inputs.

Furthermore, these works retrieve services based only on exact keyword matches whereas we define a general method to measure the similarity between services. Thus, an essential concept within our model is service similarity, and an essential issue is the measurement of two compatible service behaviors. This is an example of the widely researched nearest neighbor problem in metric spaces [42, 43] where "nearest to a given query service" is interpreted as the service with the smallest distance to an example service.

Our work introduces a model for services, and it is therefore helpful to consider other approaches to this subject. The area of service modeling is vast and mostly out of the scope of this work, and we therefore focus on a small but diverse group of formal models for *service discovery*. These are models that examine service descriptions that are informative for search, and address the need to *retrieve* a relevant service.

The most popular approach to service description is the WSDL file. These files describe service operations and parameters as elements in standard XML format. In [44] and [45], custom similarity functions are designed based on information gleaned from a statistical analysis of large WSDL collections. A bag-of-words model is used in [44] to find services that are within some edit distance to indexed WSDL files. In a similar vein, [45] recommends substitutable services based on extended string matching that incorporates tendencies found in WSDL naming conventions. Searching for service alternatives is also our goal; however, our approach builds a model from service *behavior*. In addition, we query our model by *example*; that is, find substitutions using a description of behavior rather than keywords.

The use of metrics for defining service similarity is discussed in [46, 47]. The metrics in the former work are based on the similarity of terms that are extracted from the corresponding WSDL descriptions (as well as other related terms). The latter work measures service similarity based on the commonalities in the structures of the services (e.g., the internal processes used). In neither case is the similarity derived from observable behavior.

There are different interpretations of behavior. In [48] and [49], behavior is defined as data flow between service invocations, statements of logic model these connections, and similarity is defined as Boolean combinations of predicates. In [50], graph theory is used to model behavior on the basis of the dependencies among service operations, and similarity is based on graph edit distance. In contradistinction, we represent behavior as sets of input/output observations. This approach is facilitated by instrumenting software as it runs, and then collecting invocation traces in a repository [18]. This type of information can also be gleaned from deployment logs or other histories of program executions [19].

Instead of modeling behavior, [51] presents a service retrieval approach inspired by the relational model. The goal is to optimize QoS queries using techniques from database query processing. The research in [51] assumes sets of functionally equivalent services, whereas our work seeks to *find* equivalent services. In addition, [51] emphasizes differences in QoS parameter when choosing between equivalent services, whereas we emphasize differences in behavior.

An essential concept within our model is the similarity of services. The concept of similarity, it should be pointed out, has been researched extensively in other fields, including psychology [52,53]. As services are software modules, it is worth mentioning that the notion of similarity is fundamental to many applications in software engineering, including code completion [54], extracting source code snippets [40], visualizing software interfaces [55], and indexing source code repositories [38].

The challenge we address — efficiently finding the services most similar to a given service — is an example of the widely researched nearest neighbor problem [43,56]. Our criterion for similarity is defined by metric spaces [42], where "nearest to a given service" is interpreted as the service with the smallest distance to the service. Previous applications of metric spaces to service-oriented computing focused on finding geographically close services in ubiquitous computing environments [57, 58]. The goal is to make frequent service lookups efficient by balancing the load among service locators. Similar to our approach, services in [58] are treated as points in a metric space and then embedded into a vector space. However, similarity is not the focus of that work and it assumes that an ontology is available to model service properties.

2.4 Service Substitutions

The essence of the work described here is to extract the behavior of Web services by judicious *testing*; then use the results of testing to measure service *similarity* and derive *samples* of service functionality using cluster analysis. We discuss here briefly prior research in these subjects.

Testing is a familiar means for assessing the behavior of software; usually to ascertain that the software performs according to its specification [59]. In the context of SOA, testing has been similarly used to gain confidence that the service integrating application performs as expected [60]. In this vein, [61] suggests verifying and validating Web services with techniques of data perturbation. In contradistinction, our objective here is to generate a

suite of tests that extract the behavior of services.

We use cluster analysis to determine when there are enough tests to reflect service behavior. Cluster analysis is an unsupervised learning method such that the goal is to partition data into sets based on a notion of *resemblance*: A formula that specifies a measure of similarity. This measure defines what the learning method expects to 'see' when comparing two objects [62]. There are two motivations for clustering: *data exploration* where clusters provide insight into the nature of the data (e.g., visualizations) and *data generalization* where clusters form equivalent classes (i.e., cluster membership indicates proximity). The second motivation is a form of unsupervised classification, that is, the learning method does not require class labels given by experts. In contrast, supervised methods of classification learn to recognize objects from manually labeled training data.

Generally, clustering algorithms divide into two main approaches: hierarchical and partitional [63]. Each approach has advantages and there is no definitive rationale for choosing one over the other. However the choice of algorithm depends on several aspects of the data such as type (qualitative or quantitative), dimensionality (single value or vector value), and natural cluster structure (overlapping or non-intersecting). For instance, the structure of hierarchical approaches generate nested clusters that form a tree structure and partitional approaches generate disjoint sets of clusters. If the clustering problem does not require a particular choice, then the selection decision reduces to algorithm performance or quality.

A "good" clustering is one where inter-cluster distance (i.e., cohesion) is small and intra-cluster distance (i.e., separation) is large. Other measures of validity are defined by the extent to which clusters adhere to an intended purpose. For example, classification by clustering organizes data objects into groups so that they are more efficiently retrieved later. In this case, measures of precision and recall may be more important means to evaluate the quality of clustering.

Several works have explored the subject of service clustering [41, 64–66]. Invariably, the requisite information for clustering is derived from the files that describe the Web services (WDSL). Typically, the extracted keywords are processed with information retrieval

techniques (e.g., singular vector decomposition) to create vector representations of services. The similarity of these representations is then measured with standard IR measures (e.g., cosine). In contradistinction, our methods ignore WSDL descriptions entirely, and the similarity measures that we introduce have been designed to estimate the similarity of *behaviors*, rather then the similarity of *descriptions*.

The goal of the four works discussed above is to locate services that satisfy a need; and this need is assumed to be specified with IR-style *keywords*. In [67] users must describe their needs with keywords, which are then used to search through WSDL files for appropriate matches. Somewhat different, [68] attempts to locate services based on *non-functional attributes*, such as security or reliability. Our purpose here is different from these all: We aim to locate substitutions. Hence, our search goal is specified with an *example* (not keywords): Find services that performs like a particular service.

Finally, we discuss three works that are most closely related to the work described in this paper. Finding substitutable services is also the subject of [69]. It shares an important principle with our work: Services are described with tables that specify their inputs and outputs, and table-to-table similarities are used to determine substitutability. The approach is elegant, but it ignores many critical issues, including the automatic generation of such tables with unbiased samples, the determination of minimal samples that meet an accuracy threshold, the clustering of services into equivalence sets, and the appropriate handling of null outputs of various kinds.

The subject of [70] is a process for evaluating component replaceability. It, too, adopts the approach that service behavior is observed through its input-output mappings. But, like the previously discussed paper, it has a single focus: the generation of a test suite and the analysis of its results.

Chapter 3: A Model for Information Services

We begin our investigation of the service repair problem by considering an essential issue: How to describe service functionality? Given a service, a good substitute should be compatible and should behave similarly. To this end, our model captures aspects of information services in formal structures that *accurately* represent their syntax and semantics. Thus this dissertation introduces a model for information services that emphasizes the ability to represent service functionality and recommend service substitutes.

The main contributions of this chapter are:

- In Section 3.1 and 3.2, we present a definition of services that models the syntax and semantics of services.
- Then we present two pragmatic adjustments to our model in Section 3.3: behavior table samples and exceptions.

3.1 Domain Expressions

Before defining services, we address how to model the syntax of service input and output parameters with a formal notion of *domains*. A domain is a set of values that share similar syntax and semantics. A domain is either *simple* or *complex*. A simple domain is a set of scalar values of a certain *type*. For now, we assume only two types: numbers and character strings. The values of a domain have shared semantics. For example, a domain may be a range of temperatures, a set of Zip codes, or a set of stock symbols. Simple domains may be combined to form complex domains with two operators, *aggregate* and *sequence*.

- Given a domain α, the *aggregation* of α forms a new domain, denoted $\{\alpha\}$, in which each element is a *set* of elements of α.

- Given domains $\alpha_1, ..., \alpha_n$, the *sequencing* of $\alpha_1, ..., \alpha_n$, forms a new domain, denoted $(\alpha_1, \ldots, \alpha_n)$, in which each element is a sequence of n elements in which the ith element is from the domain α_i.

For example, a set of Zip codes may be aggregated to denote a destination, $\{Zipcode\}$. As another example, values of latitude, longitude and elevation may be sequenced to denote a 3-position coordinate, $(latitude, longitude, elevation)$.[1]

These two operations may be combined to create arbitrarily complex domain expressions from simple domains. Let α denote a simple domain, then the grammar for defining domain expressions Δ is:

$$\Delta ::= \alpha \mid \{\Delta\} \mid (\Delta, \ldots, \Delta) \quad (3.1)$$

3.2 Services

We view services as software components that provide data in response to requests. These components are stateless and free of side-effects; that is, their responses depend only on the input in the requests, and these responses are the only outcome of the requests. As such, an information service is a data object encapsulated in specific input and output protocols. Alternatively, one could consider each service as a small database that is wrapped to process one type of query. Hence our model abstracts information services as functions from input domains to output domains:

Definition 3.1. *Let A and B be two domains, a service s is a function $s : A \longrightarrow B$.*

An example of a service over simple domains is the service s_1 that receives the name of a country and returns its capital city $s_1 : country \longrightarrow city$. Examples of services over complex domains are the service s_2 that receives a set of Zip codes and returns the current average temperature and humidity $s_2 : \{zip\} \longrightarrow (temp, humidity)$; the service s_3 that

[1] Note that in set-theoretical terms, the sequence of domains corresponds to their Cartesian product.

receives the title of a movie and return pairs of theaters and show times $s_3 : title \longrightarrow \{(theater, time)\}$; and the service s_4 that receives a combination of stock symbol and date and returns the opening, closing, low and high prices for that date $s_4 : (stock, date) \longrightarrow (open, close, high, low)$.

Consider a service s with input domain A and output domain B.

Definition 3.2. *The* signature *of s is the pair (A, B) of input and output domain expressions.*

Finally, two services s_1 and s_2 are said to be *compatible* if they have the same input domain and the same output domain, i.e., same signature. A repository of services $\mathcal{R}$ may thus be *partitioned* into subsets of compatible services. It should be emphasized that this partitioning is largely syntactical, and compatible services could behave very differently. For example, there could be several compatible services from the input domain *stock* to the output domain *price*. However, one would return the most recent closing price of the stock, another could return the 90-day average, and yet another could return the 5-week high. Hence a repository may contain a collection of compatible services but subsets may behave very differently.

3.2.1 Behavior

A good substitution for a given service should not only be compatible, but should also *behave* similarly. Thus in our model, the semantics of a service is provided by the extension of the function, which we call service behavior.

Definition 3.3. *The* behavior *of s is the set of values in the domain A and their matched values in the domain B: $\{(a, s(a)) \mid a \in A\}$.*

Definition 3.4. *An individual behavior pair $(a, s(a))$ is an* instance *of the service s.* [2]

[2] This distinction between signature and behavior is similar to the distinction between intension and extension, common in logic and in databases.

Behavior may be represented in a two column *behavior table*, in which each row is an instance. For example, assuming the input domain has cardinality p, the behavior of s has p instances:

S:

A	B
a_1	$s(a_1)$
$\vdots$	$\vdots$
a_p	$s(a_p)$

It is often useful to view the service s as a simple query processor of a single table that can handle one type of query: given a value $a \in A$ it retrieves $\pi_B(\sigma_{(A=a)}(S))$.

3.3 Pragmatic Adjustments

Our definition service behavior requires two pragmatic adjustments to our model so that we can recommend substitute services. The first adjustment considers the issue of very large (or even infinite) input domains that result in intractable behavior tables. For example, a service that provides the total rainfall for a location given a Zip code would have about 43,000 instances. A service that converts degrees Fahrenheit to degrees Celsius would have an infinite number of instances.[3]

3.3.1 Reducing the Dimensionality of Behavior Tables

Our definition of service behavior assumes the input and output of services is available in their entirety, and to determine the functionality of a service we must know its entire set of instances. In practice, this assumption is impractical for a variety of reasons. First, in many instances services are stored as executable code, not as tables (a simple example is the aforementioned service that converts degrees Fahrenheit to degrees Celsius). Second,

[3]For now we ignore the fact that this type of service is better stored as a formula, such as $C = (F-32)\cdot 5/9$.

even when stored in tables, these tables are “wrapped” in protocols that permit queries that access few instances at a time. Finally, even when the tables are available, their sizes may be substantial thus requiring many costly service calls.

We address this issue by considering only *samples* of behavior tables. Such samples must be statistically valid; namely, random and of sufficient size. Of course, by using samples rather than the “true” behavior tables, our methods work with *estimated* behavior. A sample (estimate) of behavior table S will be denoted $\overline{S}$. Typically, a sample of size n is obtained by a sequence of n single-instance calls to the service. In chapter six, we show how relatively small samples (of about 16 instances) may be obtained that convey satisfactorily the “traits” of a behavior.

3.3.2 Exceptions in Behavior Tables

Formally, a service should respond to inputs that are in its domain, whereas in real-world situations, services may be invoked with inputs that are outside their domains. Hence, our second pragmatic adjustment introduces a special value called *exception* which is added to every domain. When a service receives an input outside its domain (or when it receives *exception* input), it returns *exception* in each of its outputs. This implies that tables of estimated behavior may contain exceptions.

In other words, each sample point a_i might result in three types of output $s(a_i)$: (1) a valid value, (2) an error value (illegal input), or (3) a missing value (no response). These outputs are all encoded in the behavior table. Both error values and missing values are *exceptions*. A missing value reflects a service invocation without a response. It resembles a database *null* of the type *not available* [71]. In data mining applications such values are often *imputed* from other values [72]. On the other hand, an error value indicates that the input is outside the expected domain. It resembles a database *null* of the type *not applicable* [71] (and imputation is entirely unjustified). Indeed, that one service responds to an input with a proper value, whereas another issues an error message, is an indication

of service dissimilarity. [4]

Consider this small 4-instance behavior table sample in which service exceptions are denoted with —:

$\overline{S}$:

A	B
a_1	$s(a_1)$
a_2	—
—	—
a_4	$s(a_4)$

In this example, we estimate the behavior table with a behavior table sample, which includes four service instances of two valid output and two exception output, one missing value and one error value respectively.

[4]These *exception* values are similar to database *null* values.

Chapter 4: A Language for Service Compositions

In the last chapter, we provided formal definitions for *domain expressions*, a single *service*, and associated properties with services such as the service *signature* and service *behavior*. Recall that our goal is to develop methods for repairing compositions that stop functioning due to the failure of a component service, so in this chapter we address another essential issue: recommending substitutes in service compositions.

The main contributions of this chapter are:

- We introduce a language for describing service compositions in Section 4.1 and show how to express compositions in a simple language with two operations: compose and combine.
- In Section 4.2, we define the mapping from service expressions to behavior tables.

4.1 Compose and Combine Operators

Compositions are formed from individual services (which we call *basic* services), with two operators, *compose* and *combine*. The expressions that result define new *composite* services that model the flow of data among a set of basic services found in the repository. The *compose* and *combine* operators are both binary: each connects two "compatible" services to create a composite service. The *compose* operator executes services in *sequence*, whereas the *combine* operator executes services in *parallel*. Together, they represent new composite services.

Consider two services:

$$
\begin{aligned}
s_1 : A &\longrightarrow B \\
s_2 : B &\longrightarrow C
\end{aligned}
$$

Note that the co-domain of s_1 is the domain of s_2.

Definition 4.1. *The* compose*, denoted $s_1 \circ s_2$, is defined as the composition of the two functions:*

$$
\begin{aligned}
s_1 \circ s_2 : A &\longrightarrow C \\
(s_1 \circ s_2)(a) &= s_2(s_1(a))
\end{aligned}
\tag{4.1}
$$

It is often desirable to connect the outputs of two (or more) services to the inputs of another service, or the output of one service to the inputs of two (or more) services. Towards this goal, we define the *combine* operator. Assume

$$
\begin{aligned}
s_3 : A &\longrightarrow C \\
s_4 : B &\longrightarrow D
\end{aligned}
$$

Definition 4.2. *The* combine *operator, denoted (s_1, s_2), is defined as the product of the two functions:*[1]

$$
\begin{aligned}
(s_3, s_4) : (A, B) &\longrightarrow (C, D) \\
(s_3, s_4)(a, b) &= (s_3(a), s_4(b))
\end{aligned}
\tag{4.2}
$$

[1] We use (s_3, s_4) instead of the more common functional notation $s_3 \times s_4$, and we use (C, D) instead of the more common set notation $C \times D$.

To demonstrate the use of both operators together, assume another service s_5:

$$s_5 : (C, D) \longrightarrow E$$

We compose (s_3, s_4) with s_5 to create $(s_3, s_4) \circ s_5$:

$$\begin{aligned} ((s_3, s_4) \circ s_5) : (A, B) &\longrightarrow E \\ ((s_3, s_4) \circ s_5)(a, b) &= s_5(s_3(a), s_4(b)) \end{aligned} \tag{4.3}$$

The composite service $(s_3, s_4) \circ s_5$ connects the outputs s_3 and s_4 to the input of s_5.

In duality, we can use *combine* and *compose* to connect the outputs of one service to the inputs of two (or more) services. Assume services s_6, s_7 and s_8:

$$\begin{aligned} s_6 : F &\longrightarrow A \\ s_7 : G &\longrightarrow B \\ s_8 : E &\longrightarrow (F, G) \end{aligned}$$

We compose s_8 with (s_6, s_7) to create $s_8 \circ (s_6, s_7)$:

$$\begin{aligned} (s_8 \circ (s_6, s_7)) : E &\longrightarrow (A, B) \\ (s_8 \circ (s_6, s_7))(e) &= (s_6(s_8(e)), s_7(s_8(e))) \end{aligned} \tag{4.4}$$

Finally, we can compose the service described in Equation 4 with the service described in Equation 3 to create the service

$(s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5)$:

$$(s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5) : E \longrightarrow E \tag{4.5}$$
$$(s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5)(e) =$$
$$s_5(s_3(s_6(s_8(e))), s_4(s_7(s_8(e))))$$

Figure 4.1 illustrates the compositions described in Equations 1–5. In these diagrams each service is represented with a rectangle with in-coming and out-going arrows. Note that each such arrow represents multiple inputs and outputs (the arrows may be labeled with the inputs and outputs).

4.1.1 Service Expressions

The following grammar defines our language in which we generate service *expressions* with *compose* and *combine* operations.

$$
\begin{aligned}
< expression > &:= (< compose >) \;|\; (< combine >) \;|\; service \\
< compose > &:= < compose > \circ < expression > | < expression > \\
< combine > &:= < combine >, < expression > | < expression >
\end{aligned}
$$

The second and third lines create, respectively, arbitrarily long compositions and combinations of *expressions*. The first line replaces expressions by basic *services*. The first line allows, in addition, to embed chains of compositions or chains of combinations, when enclosed in parentheses. The result of our grammar are formulas that represent the connections of services in compositions.

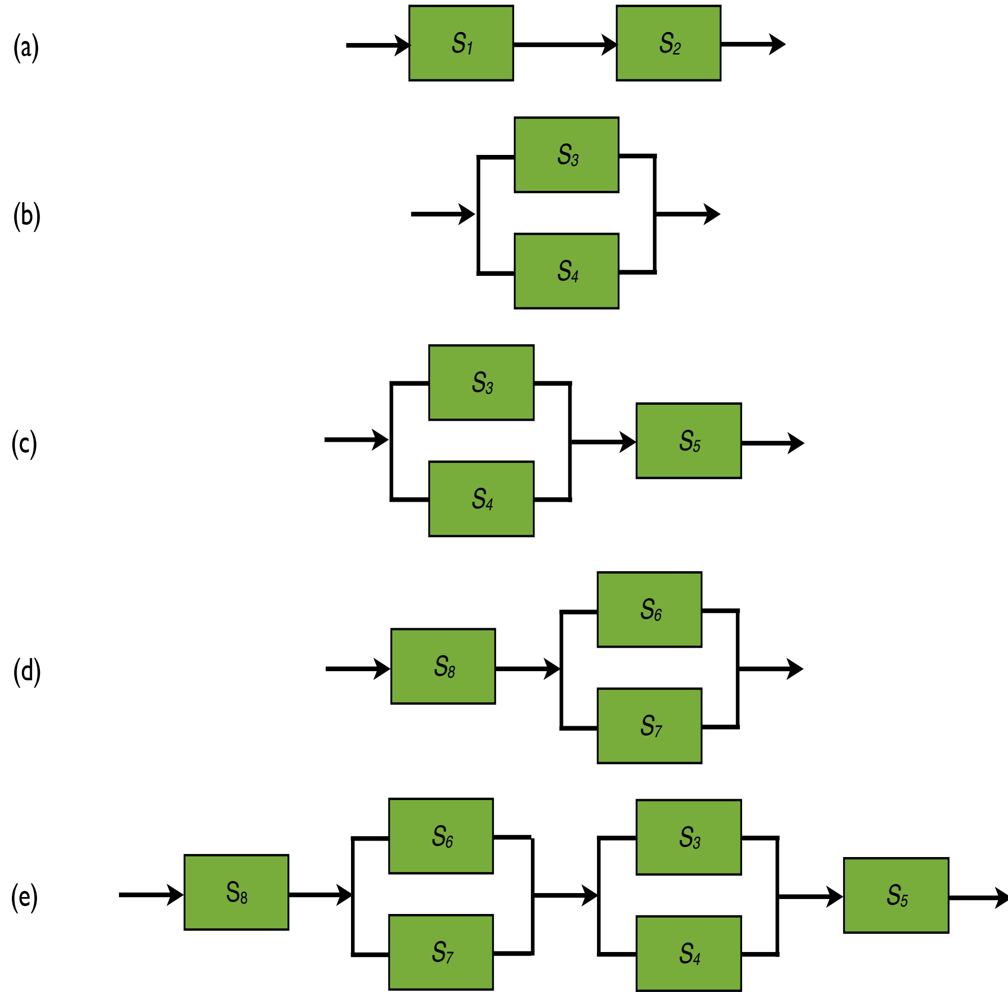


Figure 4.1: The *compose* and *combine* operators.

4.2 Behavior of Service Expressions

So far, we supposed a repository of services $\mathcal{R}$, defined two operators, and we assume services $s_1, \ldots, s_k$ in $\mathcal{R}$ have encapsulated behavior $\overline{S_1}, \ldots, \overline{S_k}$ captured in behavior table samples. Until given an interpretation, service expressions represent syntactically correct composite services but lack any meaning. We turn now to a discussion on assigning meaning to service expressions and then proceed to propagate the behavior of basic services to subexpressions.

The behavior (extension) of service compositions is derived from the behavior of basic services, as implied by Equations (1) and (2). Consider two services s_1 and s_2 with corresponding behavior tables S_1 and S_2.

1. The behavior table of their composition $s_1 \circ s_2$ (Equation (1)) is simply the relational algebra join followed by a projection that removes the join columns $\pi_{A,C}(S_1 \bowtie S_2)$.

2. The behavior table of their combination (Equation (2)) is simply the Cartesian product $S_1 \times S_2$.

However, recall our pragmatic adjustment to use only samples (estimates) of the extensions of services. This suggests that we must derive the samples of the behavior of compositions from the samples of the behavior of the basic services; that is, from $\overline{S_1}$ and $\overline{S_2}$:

$$\overline{\pi_{A,C}(S_1 \bowtie S_2)} \approx \pi_{A,C}(\overline{S_1} \bowtie \overline{S_2}) \tag{4.6}$$

$$\overline{S_1 \times S_2} \approx \overline{S_1} \times \overline{S_2} \tag{4.7}$$

The join in Equation (6) presents a pragmatic difficulty in that some of the values in the output column of $\overline{S_1}$ may not be available in the input column of $\overline{S_2}$ resulting in samples with an insufficient number of instances. This could be remedied by invoking s_2 with the unmatched values in the output column of $\overline{S_2}$. These unmatched values are preserved with the relational algebra left outer join.

As an example, consider services

$$\begin{aligned} s_1 : A &\longrightarrow B \\ s_2 : C &\longrightarrow D \\ s_3 : (B, D) &\longrightarrow E \end{aligned}$$

with the following behavior samples[2]

$\overline{S_1}$

A	B
1	2
2	3

$\overline{S_2}$

C	D
1	2
2	4
3	6
4	8

$\overline{S_3}$

B	D	E
2	2	4
2	4	6
2	6	8
3	2	5
3	4	7
3	6	9

Consider now the service composition

$$(s_1, s_2) \circ s_3 : (A, C) \quad \longrightarrow \quad E$$

The behavior sample for (s_1, s_2) is[3]

$\overline{(S_1 \times S_2)}$

A	C	B	D
1	1	2	2
1	2	2	4
1	3	2	6
1	4	2	8
2	1	3	2
2	2	3	4
2	3	3	6
2	4	3	8

and for the entire expression $(s_1, s_2) \circ s_3$

[2]Intuitively, s_1 increments its input, s_2 doubles its input, and s_3 sums its two inputs.
[3]Note that the columns have been rearranged.

$\pi_{A,C,E}((S_1 \times S_2) \bowtie S_3)$

A	C	E
1	2	4
1	2	6
1	3	8
1	4	–
2	1	5
2	2	7
2	3	9
2	4	–

Note that the outer left join of (A, C, B, D) and (B, D, E) introduces two exceptions (null values). These will require invoking s_3 twice more, with the inputs (2,8) and (3,8), to obtain the values 10 and 11, respectively.

Chapter 5: A Definition for Service Similarity

Given a service s_0 and a set of services, we are interested in the service s^* whose behavior is most similar to s_0. Thus, in this dissertation, the second issue we address is service substitution. Towards this end, we define a notion of *service similarity* that captures our intuition about comparing two behavior tables. In what follows, we describe this definition in detail and discuss how we overcome the challenges related to measuring the similarity of behavior.

The main contributions of this chapter are:

- We present in Section 5.1 our method for measuring the distance between elements of domain expressions.
- Then, we describe our definition of service similarity in Section 5.2 which we handle exceptions in behavior.

5.1 Service Similarity

A question that often arises in service-oriented software is whether two given services are "similar" to each other; i.e., whether one service could substitute for the other. To answer this question we must define service similarity. We limit our discussion here to the similarity of compatible services.

5.1.1 Similarity of Domain Elements

To measure the similarity between the elements of a domain we define a metric for that domain. A *metric* for a domain A is a function d from the product $A \times A$ to the real numbers that satisfies

1. $d(a_1, a_2) \geq 0$

2. $d(a_1, a_2) = 0 \Longleftrightarrow a_1 = a_2$

3. $d(a_1, a_2) = d(a_2, a_1)$, and

4. $d(a_1, a_2) + d(a_2, a_3) \geq d(a_1, a_3)$.

It should be noted that, at times, these four requirements may be too restrictive, and in some applications where distances need to be measured, some of the requirements are relaxed. For example, the requirement for symmetry (the third requirement) may be inappropriate when measuring driving distances between cities, because connecting roads could be different in each direction (the measure is called a *quasi metric*). In some applications, the requirement that the distance between different points be non-zero (the second requirement) is relaxed (the measure is called *pseudo metric*); for example, if we wish to measure the distance between two points in the plane based only on their first coordinate: $d((x_1, x_2), (y_1, y_2)) = \mid x_1 - y_1 \mid$. An example of relaxing the triangular inequality (the fourth requirement) is in distances based on the co-occurrence of terms in documents (the measure is called a *semi metric*). Noting that the distances here denote *behavior similarity*, it follows that all four requirements are sensible. For example, substituting a service s_1 with s_2 and then substituting s_2 with s_3 should not be any better than substituting s_1 with s_3 directly. Note, however, that in Section 7.1 we consider distances that are not symmetric.

Our approach is to establish distance metrics for simple domains, and then extend these metrics to complex domain expressions.[1]

In this dissertation, we focus on two simple domains: real numbers R and character strings S. We note that extending the work to other domains should not be difficult. Distances among real numbers will be measured with the *absolute value metric*, and distances among strings will be measured with the *Levenshtein edit distance*. These two metrics are extended to metrics for general domains as follows.

[1] Note that metrics measure distance, and eventually we shall convert distance to similarity.

Let x, y be elements of domain α and assume that $\alpha = (\alpha_1, \ldots, \alpha_n)$; that is, the final domain operation to create α was sequencing. Then x and y are sequences: $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. We define the distance between x and y as the *sum* of the distances between the sequence components:[2]

$$d(x, y) = \sum_{i=1}^{n} d(x_i, y_i) \tag{5.1}$$

Let x, y be elements of domain α and assume that $\alpha = \{\alpha_1\}$; that is, the final domain operation to create α was aggregation. Then x and y are sets: $x = \{x_1, \ldots, x_n\}$ and $y = \{y_1, \ldots, y_m\}$. We define the distance between x and y as the Hausdorff distance between the sets. That is, first we calculate the directed distance from x to y: For each element in x we choose the smallest of its distances to the m elements of y, and then we choose the largest of the n distances thus obtained; we similarly calculate the directed distance from y to x; and finally — to assure symmetry — we adopt the highest of the two distances:

$$d(x, y) = \max\{\max_{x_i \in x}\{\min_{y_j \in y}\{d(x_i, y_j)\}\}, \max_{y_j \in y}\{\min_{x_i \in x}\{d(y_j, x_i)\}\}\} \tag{5.2}$$

As the domains α_i used to create α could themselves be complex, these definitions are reapplied recursively, until all distances are calculated in simple domains, using the two basic metrics.

To illustrate, assume a domain $\alpha = (S, \{R\})$ and two values from this domain: ('Jack', $\{1, 3, 5\}$) and ('Jill', $\{5, 8, 11, 14\}$). The Levenshtein distance between 'Jack' and 'Jill' is 3; the Hausdorff distance between $\{1, 3, 5\}$ and $\{5, 8, 11, 14\}$ is 9; and the overall distance between ('Jack', $\{1, 3, 5\}$) and ('Jill', $\{5, 8, 11, 14\}$) is therefore $3 + 9 = 12$.

Recall, our approach requires a corresponding distance function for each domain expression operator. The distance functions selected above are simple but not arbitrary. For example, an aggregation uses the maximal pair-wise distance rather than the average. We

[2]Note, we assume all component distances are normalized to the same range before summation.

took this cautious approach to distance calculation since we do not want to understate the difference between two services. We could allow for other distance functions to be used; however, regardless of the chosen distance function, we must ensure the final distance calculation remains a metric. Note, the eventual distance function on α thus defined is indeed a metric. This follows from (1) the distances defined on the simple domains (absolute value and Levenshtein) are both metrics, (2) the distance between sequences as defined in Equation 5.1 is a metric,[3] and (3) the distance between aggregates as defined in Equation 5.2 is a metric.[4]

Finally, we define the *similarity* between two elements of a domain as the *reciprocal* of the distance:

$$sim(x_1, x_2) = 1/d(x_1, x_2) \tag{5.3}$$

5.1.2 Similarity of Behavior

Recall that we interpreted the similarity of two services (of the same signature) as the similarity of their behaviors. Assume two services with identical signatures:

$$\begin{aligned} s_1 : A &\longrightarrow B \\ s_2 : A &\longrightarrow B \end{aligned}$$

The similarity between s_1 and s_2 is defined as the *average* of the similarity of their outputs for all possible inputs. Let $|A| = n$. Then:

$$sim(s_1, s_2) = \frac{1}{n} \sum_{a \in A} sim(s_1(a), s_2(a)) \tag{5.4}$$

[3] This metric is termed the *product metric*, as the new space is a product of the component spaces.

[4] The final step in that process, taking the maximum of the two directed distances, assures the required symmetry.

As an example, assume a small domain $A = \{1, 2, 3, 4\}$ with the absolute value metric, a service s_1 that multiplies its input by 5, a service s_2 that multiples its input by 6, and a service s_3 that adds 10 to its input. Their behavior tables can be combined:

A	s_1	s_2	s_3
1	5	6	11
2	10	12	12
3	15	18	13
4	20	24	14

To calculate the similarity of any two services we average the similarity of their outputs for the four possible inputs: $sim(s_1, s_2) = 1/2.5 = 0.4$, $sim(s_1, s_3) = 1/4 = 0.25$ and $sim(s_2, s_3) = 1/5 = 0.2$.

5.1.3 Service Exceptions

As we mention in chapter three, similar to null values in database tables, behavior tables may have missing values as well. This happens when the service cannot deliver an output value for a particular value of the input. These missing values are termed *service exceptions*. Consider this small example in which service exceptions are denoted with —:

A	s_1	s_2
1	b_1	c_1
2	b_2	—
3	—	c_3
4	b_4	c_4
5	b_5	—
6	b_6	c_6
7	—	—

For calculating distances and similarities in the presence of exceptions, our approach is to substitute each distance that cannot be calculated due to an exception with the *maximal* known distance, again we adopt a cautious approach that prefers to understate the similarity than to overstate it. In the example,

$$d(b_2, -) = d(-, c_3) = d(b_5, -) = d(-, -) = \max\{d(b_1, c_1), d(b_4, c_4), d(b_6, c_6)\}$$

With distances now available, similarities are obtained by reciprocity, and the overall service similarity by average:

$$sim(s_1, s_2) = \frac{1}{7} \cdot (sim(b_1, c_1) + sim(b_4, c_4) + sim(b_6, c_6) + 4 \cdot \min\{sim(b_1, c_1), sim(b_4, c_4), sim(b_6, c_6)\})$$

Denote A_0 the subset of A in which both services do not have exceptions. Let $|A| = n$ and $|A_0| = n_0$. Therefore the number of instances in which at least one service has an exception is $n - n_0$. Then

$$sim(s_1, s_2) = \frac{1}{n} \cdot (\sum_{a \in A_0} sim(s_1(a), s_2(a)) + (n - n_0) \cdot \min_{a \in A_0}\{sim(s_1(a), s_2(a))\}) \tag{5.5}$$

Obviously, in the absence of exceptions, Equation 5.5 reduces to Equation 5.4. A shortcoming of this equation is that when the number of valid similarities n_0 is small, the measurement of service similarity is less robust. In an extreme case, we might even have $n_0 = 0$; that is, there are no valid similarities and $sim(s_1, s_2)$ is not well-defined. This could happen

when on each value of the input domain one of the services reports an *exception*, or when one of the services is entirely inoperative. Preferring a cautious approach, when one value is *exception*, we define:

$$d(a, -) = \max\{d(a, x) \mid x \in B\}$$

.

When both values are *exception*:

$$d(-, -) = \max\{d(x, y) \mid x, y \in B\}$$

5.2 Directed Similarity

Our samples are obtained by individual requests to the service, and when a service request results in an exception, we enter an "exception" into the behavior table. However, keeping in mind that our main application is service substitution, our calculation of service similarity (behavior similarity) would no longer be symmetric.

When considering a substitution of s_1 with s_2, the similarity calculation will ignore all instances in which s_1 has an exception in its behavior; and when considering a substitution of s_2 with s_1, our calculation will ignore all instances in which s_2 has an exception in its behavior. The reason is that we assume that the service to be replaced is operating correctly, and its exceptions were a result of illegal out-of-range requests. We expect the "new" service to deliver valid output only in the instances in which the "old" service delivered valid output.

Denote A_1 the subset of A in which s_1 does not have exceptions and denote A_2 the subset of A in which s_2 does not have exceptions. Recalling that A_0 is the subset for which both services do not have exceptions, we have $A_1 \cap A_2 = A_0$.

Let $|A_1| = n_1$ and $|A_2| = n_2$. Therefore the number of instances in which s_2 has exception but s_1 does not is $n_1 - n_0$. The *directed similarity* from s_1 to s_2 is

$$\vec{sim}(s_1, s_2) = \frac{1}{n_1} \cdot (\sum_{a \in A_0} sim(s_1(a), s_2(a)) \\ + (n_1 - n_0) \cdot \min_{a \in A_0} \{sim(s_1(a), s_2(a))\}) \tag{5.6}$$

Observe that $\vec{sim}(s_1, s_2)$ in Equation 5.6 is derived from $sim(s_1, s_2)$ in Equation 5.5, by substituting n_1 for n in two places.

In the previous example, when s_2 is considered a substitute for s_1 the directed similarity would be:

$$\vec{sim}(s_1, s_2) = \frac{1}{5} \cdot (sim(b_1, c_1) + sim(b_4, c_4) + sim(b_6, c_6) \\ + 2 \cdot \min\{sim(b_1, c_1), sim(b_4, c_4), sim(b_6, c_6)\})$$

whereas when s_1 is considered a substitute for s_2 the directed similarity would be:

$$\vec{sim}(s_2, s_1) = \frac{1}{4} \cdot (sim(b_1, c_1) + sim(b_4, c_4) + sim(b_6, c_6) \\ + \min\{sim(b_1, c_1), sim(b_4, c_4), sim(b_6, c_6)\})$$

5.3 Similarity of Services with Different Outputs

The definition presented so far made a simplifying assumption, which we shall now attempt to relax. Recall that the signature of a service s is the pair (A, B) of its input and output domain expressions. Our discussion of service similarity was limited to services with identical signatures; that is, services with identically-structured inputs and outputs. This assumption allowed us to compare two services s_1 and s_2 that have the same signature (A, B), by considering triples: an element of domain A, the element of domain B assigned

to it by s_1, and the element of domain B assigned to it by s_2. The distance between the latter two is calculated with the domain metric, and then used in the calculation of the similarity between s_1 and s_2.

This assumption implies that we cannot assess the similarity of two weather services when one provides the temperature at a given location, whereas the other provides both the temperature and the barometric pressure. Possibly, simply ignoring the second output could yield a service that is more satisfactory than the stand-by services that provide temperature only.

While substituting a given service with a service that has a different *input* domain could be useful at times (e.g., in some situations a service that expects a Zip code could be substituted with a service that expects the name of a municipality), it would be rather difficult to calculate the similarity of such services, as there would be no common points for comparison. We therefore continue to assume that services have the same input domain.

Assume now services s_1 and s_2 with signatures (A, B_1) and (A, B_2). Our approach is to extend the definition of distances among elements of the same domain (as given in Section 5.1) to elements of different domains. Once this distance has been defined, we shall continue as before: Define similarity of elements by reciprocity, and similarity of services as the average similarity of outputs. The definition of domain expressions in Equation 3.1 allowed arbitrarily complex expressions to be created from simple domains using aggregation and sequencing. Measuring the distance among elements of arbitrarily different domains would be rather complex and of limited practicality. This is because distances are likely to be high and the plausibility that one service could substitute for another is likely to be very low. We therefore limit our discussion to B_1 and B_2 of three structures:

1. A sequence of simple domains: $(B_1, \ldots, B_n)$. Each element of the resulting domain is a sequence of scalars; for example: temperature, humidity and barometric pressure.

2. An aggregate of a sequence of simple domains: $\{(B_1, \ldots, B_n)\}$. Each element of the resulting domain is a table; for example, a set of triples each describing a gas station

with longitude, latitude, and the oil company.

3. A combination of both; that is, a sequence of simple domains and an aggregate of a sequence of simple domains (in either order): $((B_1, \ldots, B_n), \{(B_{n+1}, \ldots, B_m)\})$. Each element of the resulting domain is a sequence and a table; for example, a portfolio of stocks (each described with stock symbol, price and quantity), with a date and the tax-id of its owner.

For brevity, we refer to these structures as sequence, table, and sequence+table. Measuring distances among elements of these structures involves six cases: (1) sequence *vs.* sequence, (2) table *vs.* table, (3) sequence+table *vs.* sequence+table, (4) sequence *vs.* table, (5) table *vs.* sequence+table, and (6) sequence *vs.* sequence+table.

Case 1: Assume B_1 and B_2 are both sequences of simple domains. Denote $(B_1, \ldots, B_n)$ the domains that are in both B_1 and B_2, $(C_1, \ldots, C_p)$ the domains that are in B_1 but not in B_2, and $(D_1, \ldots, D_q)$ the domains that are in B_2 but not B_1. That is,

$$\begin{aligned} B_1 &= (B_1, \ldots, B_n, C_1, \ldots, C_p) \\ B_2 &= (B_1, \ldots, B_n, D_1, \ldots, D_q) \end{aligned}$$

Now assume $x_1 \in B_1$ and $x_2 \in B_2$,

$$\begin{aligned} x_1 &= (b_1, \ldots, b_n, c_1, \ldots, c_p) \\ x_2 &= (b'_1, \ldots, b'_n, d_1, \ldots, d_q) \end{aligned}$$

To measure the distance from x_1 to x_2, we remove the subsequence $(d_1, \ldots, d_q)$ in x_2 and append a sequence of p *exception* values. To measure the distance from x_2 to x_1, we remove the subsequence $(c_1, \ldots, c_p)$ in x_1 and append a sequence of q *exception* values. If either $(d_1, \ldots, d_q)$ or $(c_1, \ldots, c_p)$ are empty (that is, one sequence is contained in the other), we only append the necessary sequence of exceptions. In the special case when one of the

sequences is *empty*,

$$B_1 = (C_1, \ldots, C_p)$$
$$B_2 = ()$$

we proceed according to the same procedure appending a sequence of p exceptions to B_2. In each case, the measurement of distances in the presence of exceptions is done as described in Section 5.1.

Case 2: To measure the distance between two different tables, we apply the usual procedure described in Section 5.1. This involves measuring the distance between every row of one table and every row of the other table, and then calculating the Hausdorff metric. The rows are of differently structured tables, but such distances were discussed in Case 1. A special case that needs to be considered is when one of the tables is empty. For example, $B_1 = \{(B_1, \ldots, B_n)\}$, and B_2 is empty. In this case we consider B_2 to be a table with a single row of n exceptions, and proceed with the calculation of the Hausdorff metric.

Case 3: To measure the distance between table+sequence and table+sequence, we, again, follow the usual procedure. This requires totaling the distance between the sequences (as in Case 1) and the distance between the tables (as in Case 2).

Case 4: To measure the distance between a sequence and a table, we consider the sequence as a table with one row, and proceed as in Case 2.

Case 5: To measure the distance between a table and a sequence+table, we convert this case to Case 3 by adding an empty sequence to the former. The distance is then the sum of the distance of the sequences and the distance of the tables. Recall that the distance between an empty sequence and a sequence was discussed in Case 1.

Case 6: To measure the distance between a sequence and a sequence+table, we, again, convert this case to Case 3, this time by adding an empty table to the former. The distance between an empty table and a table was discussed in Case 2.

As a simple example, assume that the common input domain is Zip code and the output domains are both sequences of simple domains: $B_1 =$ (Temperature, Humidity) and $B_2 =$(Temperature, Pressure). To measure the directed similarity $\vec{sim}(s_1, s_2)$, we modify the behavior table of s_2 to remove Pressure and insert a column Humidity with *exception* in every row, and proceed with the standard calculation of distances in the presence of exceptions. Similarly, to measure the directed similarity $\vec{sim}(s_2, s_1)$ we modify the behavior table of s_1 to remove the column Humidity and insert a column Temperature with *exception* in every row.

5.3.1 Domain Alignment

Consider a weather service that outputs three different temperatures: current, minimal, and maximal. An alternative service might provide the same three temperatures, but in a different order: minimal, current, maximal. Since all 6 outputs are of the same domain, and lacking any further knowledge, our methods would rely on the given order and would pair the outputs incorrectly, with negative impact on the resulting similarity. A similar issue arises with input domains.

A "brute force" approach to this problem is to calculate service similarities under all possible alignments and adopt the most favorable. While this sounds combinatorially high, the numbers involved are relatively low.

As an example, consider services s_1 and s_2 both with signature

$$((A, A, B, B)\{(C, C, D, D)\}, (X, X, Y, Y))$$

that is, both services receive a sequence of four values and a table of four columns and return a sequence of four values. Since there are 6 pairs of identical domains, altogether there are $2^6 = 64$ different ways in which their domains could be aligned.

To reduce the complexity we may use heuristic approaches that would find good substitutions, though not necessarily the best. For example, a simple heuristic is annotate

domains with their distributions (as extracted from their behavior table). In case of numeric domains, this could be simply the average.

In the example, assume that A is a numeric domain, and assume that in s_1's behavior table the average value of A is 20 in its first appearance and 60 in its second, whereas in s_2's behavior table its average is 48 and 10, respectively. This suggests calculating the similarity between the services with the A columns switched in one of the services; the other similarity will probably be lower.

Chapter 6: A Heuristic for Sampling Small Behavior Tables

In the chapter three, we discussed a pragmatic adjustment to our model where we assume samples of service behavior could be stored in a repository. Next, we describe a method to extract a representative sample of behavior from service calls. In what follows, we describe a heuristic to extract samples of behavior while methodically learning a sample size.

The main contributions of this chapter are:

- In Section 6.1, we present a method to extract a sample of service behavior by means of testing and then describe our heuristic for detecting when we have enough service instances.
- Then, we present the results of our experiments that indicate that our approach is viable, with cluster accuracy exceeding 75% using as few as 16 sample tests points in Section 6.2.
- Additionally in Section 6.2, statistical significance tests validate that the resulting service clusters are representative of service behavior and do not occur by chance.

6.1 Approach

Our heuristic involves two principal tasks: *eliciting* service behavior and *clustering* services according to that behavior. Service elicitation begins with the partitioning of the service repository according to signatures. Then, each compatible service is tested with a common sample of test points, and the responses are saved in *behavior tables samples*. Clustering begins with the measurement of similarity between every two behavior table samples in each signature.

An important objective of the heuristic is to use a low number of tests to generate service behavior of high quality. To this end, we repeat these two tasks, each time incrementing the number of sample tests geometrically, until *convergence* is reached. Convergence occurs when the difference between successive clusterings is small. After convergence, behavior table samples are saved to the service repository. Note, however, the clusters themselves are not saved. Indeed, the clusters are simply a diagnostic to gauge the quality of behavior tables samples, and not for making predictions. Figure 6.1 illustrates this the steps of this heuristic.

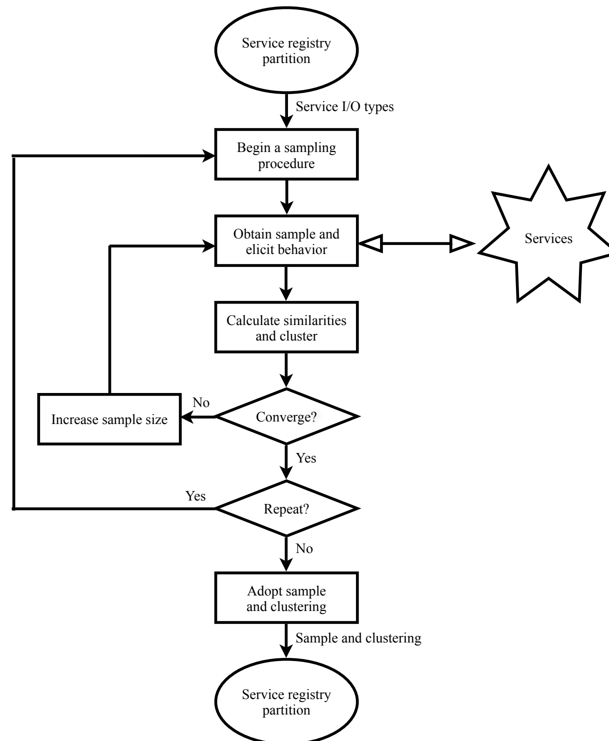


Figure 6.1: Heuristic for small samples of behavior

We begin by formalizing the environment in which we operate. We assume a service

repository with a potentially large number of registered services. The repository has a list of available domains. When a service is registered, the submitter is requested to associate a domain with each of the inputs and outputs of the service (the list of domains can be expanded if necessary).

Each service is provided with a *wrapper*. The wrapper is used to interface with the service: It processes the user input, formats it to fit the service requirements, invokes the service, and formats the service output in a behavior table (as if it were an answer from a relational database).

6.1.1 Partitioning Services by Their I/O Domains

Recall, we assume that all the services have been pre-classified into signatures based on their input and output domains. That is, the services in each signature have the same number of inputs, the same number of outputs, and their inputs and outputs are associated with the same domains. Obviously, the commonalities among services in the same signature are more syntactical than semantical.

6.1.2 Eliciting Service Behavior Tables

Consider a set of n compatible services. Our first step in clustering this signature is to obtain a common set of input values $d_1, d_2, \ldots d_p$ for testing its services. The test points d_i are determined by *sampling* the declared domain *randomly*. Each service is then invoked with these test points, returning output values $r_1, r_2, \ldots, r_p$. Note that d_i and r_j could be any domain expression.

The result of all tests is a behavior table sample that denotes the *behavior* of the service, as shown in Table 6.1.

Table 6.1: Service Behavior

d_1	r_1
d_2	r_2
$\vdots$	
d_p	r_p

Selecting Unbiased Random Sample of Inputs

As already mentioned, the repository includes a list of domains. For accurate sampling, additional information must be available on each domain and its distribution. The service repository represents domains either as numeric ranges (e.g., temperatures or salaries), or as enumerated sets (e.g., Zip codes or stock symbols). Very large discrete domains (e.g., last names or movie titles) are represented with sufficiently large samples. In addition, information is kept on the *distribution* of domains, to be used as sampling guidelines for minimizing *selection bias*. Although missing values may be imputed, when the sample test includes a high proportion of such values, the sample is rejected and a new sample is obtained. This is intended to minimize *non-response bias*.

This phase ends when all compatible services are tested, resulting in n behavior table samples.

6.1.3 Comparing the Services

To cluster a signature of services, a similarity measure must be adopted. As we describe in chapter five, we need a measure that would calculate the similarity between two behavior tables. To summarize, consider two services S_1 and S_2. The i'th row of each table is the output of the service for the test point d_i. Let the second column of that row in each of the tables be r_i^1 and r_i^2, respectively. We define the similarity of the tables in two steps. First, we define a similarity measure $\phi(r_i^1, r_i^2)$ between two corresponding results for the same test point d_i; then, we define the overall similarity $\Phi(S_1, S_2)$ of the service behaviors as the *average* similarity of the corresponding results for each of the test points (p is number of

test input values):

$$\boxed{\Phi(S_1, S_2) = 1/p\, \Sigma_{i=1}^{p} \phi(r_i^1, r_i^2)} \quad (6.1)$$

As we described in chapter five, the definition of ϕ must cope with three difficulties. First, r_i could be a value from any domain expression; second, simple domains of each r_i could be numbers, dates, strings, and so on; finally, in addition to proper values, r_i could also be error values and missing values. Therefore, ϕ should gauge the similarity of outputs when both services respond, but also consider the number of inputs which one service can handle whereas the other cannot.

We emphasize that, as is common in information retrieval systems and search engines, the definition of ϕ could be subject to continuous evolution and fine-tuning.

Choosing the Clustering Algorithm

Since our model must deal with different types of data, we do not adopt a single clustering algorithm. The experiments described in Section 6.2 choose between two clustering algorithms: Hierarchical Agglomerative Clustering (HAC) and k-Means partitional clustering. Both algorithms are extensible; that is, their clustering can be amended when the set of elements changes. This is important for our model as we expect service repositories to be dynamic, with new services added and existing services modified or removed.

6.1.4 Determining Minimal Test Cardinality

At this point, we have clustered the services in the signature according to their behavior. Yet, we have no assurance that our clustering is accurate. Obviously, with larger tests better clustering can be expected, but excessive testing incurs both costs and risks, so we should avoid tests that are unnecessarily large. In other words, for each partition we must determine an "ideal" test size: the *smallest* number of tests that will *exceed a predetermined threshold of clustering accuracy.*

Detecting Convergence

Our heuristic is to *iterate* the process of testing and clustering the services in a partition with increased test sizes, until the clustering *converges*; that is, additional increases in test size do not yield significant changes in the clustering. This technique is often referred to as *progressive sampling* [73, 74].

To illustrate, suppose we attempt test sizes $p_1, p_2, p_3, \ldots$, and beginning with p_k the derived clusterings are identical. We can then determine that there is no need to conduct tests that are larger than p_k: The clustering *converges* at p_k.

In practice, we do not seek a point beyond which the clusterings are *identical*, but merely a point beyond which their differences are sufficiently small, and we measure differences between successive clusterings with the well-known *Rand index* [75]. Consider two clusterings C_1 and C_2 of n elements. Intuitively, this index considers all $\binom{n}{2}$ pairs of elements and calculates the proportion of "agreements" among the two clusterings on their placement. An "agreement" is placing the elements of a pair in the same cluster or in two different clusters in *both* C_1 and C_2. The index generates values between 0 (C_1 and C_2 are in complete disagreement) and 1 (C_1 and C_2 are identical).

A sequence of k clusterings will produce $k - 1$ Rand index values. Theoretically, a high value could be followed by a low value; i.e., a clustering is very similar to its predecessor, but very different from its successor. In practice, however, this rarely happens, and high values are usually followed by high values; i.e., the plot of Rand index values reaches a "plateau". Hence it should be sufficient to get a high value (two similar clusterings in succession) and then adopt the sample size that was associated with the first of these clusterings. We adopt a Rand index value α that is considered sufficiently high.

Calculating Minimal Sample Size

For statistical robustness, this entire procedure is repeated n times. At each sample size, we calculate the number of times that the clustering converged successfully (i.e., the number of times in the n experiments that the Rand index exceeded α). We adopt a threshold

value β for the minimally acceptable rate of successful convergence. Finally, when testing ends, we choose the *lowest sample size* in which the successful convergence rate exceeded β. The corresponding behavior table samples are stored in the service repository to be used in future substitute recommendations.

6.2 Evaluation

For this experiment we focused on two signatures of services.

1. Weather services that receive a Zip code and return a numeric value. $s_1 : Z \longrightarrow R$
2. Stock quote services that receive a stock symbol and return a pair of numeric values. $s_2 : S \longrightarrow (R, R)$

Notice that input domains are rather specific: They are assumed to be included in the list of domains of the service repository. On the other hand, we assumed that the output domains are generic, which adds the challenge of output matching.

6.2.1 Classifying Weather Services

For the first experiment we considered 11 weather services of 4 different groups of behavior: Four services return the temperature in degrees Fahrenheit, one returns the temperature in degrees Celsius, three return the relative humidity, and three return the barometric pressure. Our goal is to identify these clusters in the given set of services. The input domain was fully enumerated (all 27,238 Zip codes), and the sampling principle was random (without replacement): Each value in the domain had equal likelihood of being selected.

Our progressive sampling started with a sample size of 2 and doubled it repeatedly to 4, 8, 16, 32 and 64. We used the following metric: The distance between two values is their absolute difference.

Given these similarity measurements, we clustered the services using both HAC and k-Means partitional clustering. We repeated this process 6 times with increasing sample sizes, computing 5 Rand index values for each clustering method (between the clustering

obtained at 4 and the clustering obtained at 2; between the clustering obtained at 8 and the clustering obtained at 4, and so on). Our termination condition was set to be a Rand index in excess of $\alpha = 0.7$ (if that threshold is not met, then the clustering of the final test is adopted).

For added statistical significance, each of the experiments was replicated $n = 40$ times, and the minimal proportion of successful convergences was set at $\beta = 0.8$.

Table 6.2: Weather: Rand Convergence

Sample	k-Means	HAC
2	0	0
4	0.5	0.23
8	0.73	0.55
16	0.95	0.73
32	0.95	0.9
64	1	0.95

Results

Table 6.2 summarizes the results of this experiment. The columns k-Means and HAC show, for each clustering method, the percentage of experiments where the Rand index exceeded the 0.7 convergence threshold. As can be observed, for k-Means clustering, to meet the threshold of 80% successful convergence, a sample of size 16 should be used (the rate of successful convergence is then 95%). For HAC, a sample size of 32 would be necessary to meet the threshold of 80% successful convergence (the rate of successful convergence is then 90%). This suggests that for this signature of services, behavior tables samples of size 16 should be preferred.

Validation

Any clustering algorithm will produce a set of clusters, even in random data. Hence it is important to validate that our service clusters characterize the data accurately. Towards this end, we measured the *accuracy* of the cluster assignments (i.e., k-Means clustering with a sample of 16 tests), by comparing it to the true clustering (as defined by experts). Each of the two clusterings was represented in symmetric matrix in which each service is represented by a row and column, and a cell (i, j) is 1 if the services in row i and column j are in the same cluster; otherwise it is 0. High correlation between the two matrices indicates high accuracy [76]. The average accuracy in the 40 trials that were performed was 75%.

The cluster accuracy score thus obtained should be evaluated for its statistical significance. Cluster accuracy is valid only if it is unusual; that is, the clusters correspond to service behavior better than by chance. The significance of the accuracy score is tested using a statistical hypothesis test. The null hypothesis states that the cluster assignments are random (and the accuracy was obtained by chance). The alternative hypothesis states that the cluster assignments are not random (and the accuracy could not have been obtained by chance) [76]. We generated 300 cluster assignments at random. Each was compared to the matrix of the true clustering and the accuracy score was computed. Figure 6.2 shows the distribution of these random accuracy scores. Next, we compared our accuracy to this distribution to decide if it is unusual (i.e., greater than a critical value). In this case 99% of the accuracy scores were less than 0.5; hence the accuracy score of 0.75 is statistically significant.

6.2.2 Classifying Stock Quote Services

For the second experiment we considered 12 stock quote services of 4 different groups of behavior: Two services return the opening and closing prices of the stock, four return the

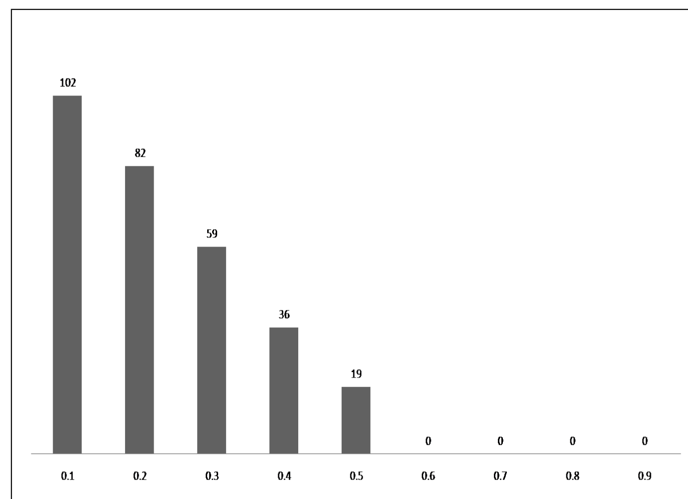


Figure 6.2: Weather: accuracy distribution

daily change and the market capitalization, four return the current price and the earnings per share, and two return the price per share and the traded volume. Again, our goal is to identify the smallest behavior tables that reflects service functionality in a set of compatible services. The input domain was assumed to be comprehensive, though not necessarily exhaustive (2,809 stock symbols), and the sampling principle was again random (without replacement): Each value in the domain had equal likelihood of being selected.

Our experiment followed the same procedure as the first experiment, with the exception of the similarity measure ϕ, which used Euclidean distance to measure the difference between two results (as each result is now a *pair* of values).

Results

Table 6.3 summarizes the results of the second experiment. As can be observed, for k-Means clustering, to meet the threshold of 80% successful convergence, a sample of size 16 should be used (the rate of successful convergence is then 83%). For hierarchical agglomerative clustering, a sample size of 32 would be necessary to meet the threshold of 80% successful convergence (the rate of successful convergence is then 80%).

Table 6.3: Stock: Rand Convergence

Sample	k-Means	HAC
2	0	0
4	0.58	0.28
8	0.75	0.45
16	0.83	0.7
32	0.85	0.8
64	0.98	0.98

Validation

The average accuracy of the clustering proposed by this heuristic (k-Means clustering with samples of size 16), as measured by the Rand index, was 84%. The significance of this accuracy score was validated with the same hypothesis test described earlier. As evident in Figure 6.3, this accuracy score is significant.

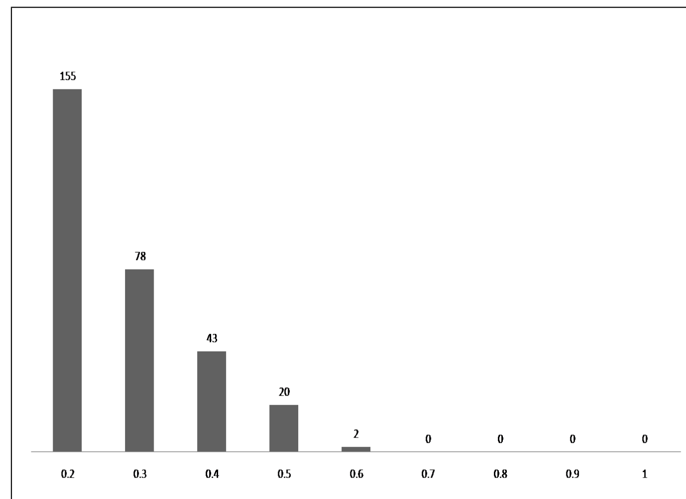


Figure 6.3: Stock: accuracy distribution

We note that in both experiments k-Means clustering proved superior. This may be due to the fact that in both cases the output of the services is quantitative. Altogether, the results

of these preliminary tests are promising, showing that our heuristic is capable of providing accurate clusters (75% and 84%) with a surprisingly low number of tests (in each case, 16 service invocations).

Chapter 7: A Heuristic for Improving Substitution Performance

In chapters three and five, we defined a model for services and a definition of service similarity. Next, we return to the issue of service substitution and offer a second heuristic for improving the performance of our recommendations. Recall that, given a service s_0 and a repository of services $\mathcal{R}$, the problem is to find the service in $\mathcal{R}$ that is most similar to s_0. This *nearest neighbor* problem arises frequently in service-oriented software architectures, when one of the services in the architecture fails and needs to be replaced by the most similar service from a repository of available services. Ideally, this substitution would be done "automatically" (i.e., without programmer's intervention) using the most similar service. In practice, however, it is safer to allow the programmer to choose from several substitution alternatives, which is a *k-nearest neighbor* problem.

The main contributions of this chapter are:

- Our approach to the issue of service substitutions is described in Section 7.1.
- In Section 7.2, we detail and analyze experiments that validate our heuristic.

7.1 Approach

We begin with a naïve treatment of this k-nearest neighbor problem, and we then offer a more efficient solution.

7.1.1 Exhaustive Search

As mentioned earlier, the service substitution issues assumes a given service s_0 and a signature of services $\mathcal{S}$, and finds the k services in $\mathcal{S}$ that are most similar to s_0; that is, services

$s_1, \ldots, s_k \in \mathcal{S}$, such that for $1 \leq i < k-1$ $sim(s_0, s_i) \geq sim(s_0, s_{i+1})$ and for every other service $s \in \mathcal{S}$ $sim(s_0, s) \leq sim(s, s_k)$.

The simplest solution, of course, is to conduct an exhaustive search. That is, calculate the similarity between s_0 and each service in the signature $\mathcal{S}$, while at each point maintaining a list of the top k services discovered so far.

Assume $|\mathcal{S}| = n$. In practice, we pre-compute all n^2 similarities and store them in an $n \times n$ matrix. When it is necessary to substitute a service s_j, the j'th row in the matrix is retrieved and top k values are extracted. The positions of these values point to the best substitutions. Note, however, that even the best substitutions could be too dissimilar to s_j, so in practice, we should define a *threshold* to aid the programmer in interpreting these substitution recommendations.

This "brute-force" algorithm has complexity $O(n^2)$. Note, however, that calculating each of the similarities, especially for services over complex domains, could become costly. Our next method attempts to reduce this cost.

7.1.2 Metric Space Embedding

A common method to reduce the effort of calculating distances (or similarities) between elements in a given metric space, is to embed the elements in another metric space, so that the distances between the mapped elements are similar to the original distances, but are (hopefully) considerably simpler to calculate [77]. Formally, let ψ be an embedding of the metric space (A, d) into the metric space (A', d'). We want to find an embedding ψ such that for any two elements $a_1, a_2 \in A$:

$$d(a_1, a_2) \approx d'(\psi(a_1), \psi(a_2))$$

where $\approx$ denotes approximation.

Let (A, d) be a metric space. We extend the metric d to measure the distance between

an element $a \in A$ and a subset $X \subseteq A$ as follows:

$$d(a, X) = \min_{x \in X}\{d(a, x)\}$$

That is, the distance between an element and a set is the minimum distance between the element and any element in the set. Now let $X_1, X_2, \ldots, X_m$ be subsets of A. We map every element a to a *vector* of its distances to these subsets:

$$\psi : a \longrightarrow (d(a, X_1), d(a, X_2), \ldots, d(a, X_m))$$

Such a mapping is called a *Lipschitz embedding*, and $X_1, X_2, \ldots, X_m$ are its *reference sets*. The intuition is that the distance between two elements a_1 and a_2 will be captured adequately by the distance between their embedded vectors:

$$\begin{aligned} d(a_1, a_2) &\approx d'(\psi(a_1), \psi(a_2)) \\ &= d'((d(a_1, X_1), d(a_1, X_2), \ldots, d(a_1, X_m)), \\ &\qquad (d(a_2, X_1), d(a_2, X_2), \ldots, d(a_2, X_m))) \end{aligned}$$

A particular version of a Lipschitz embedding commonly used in situations similar to ours selects references that are *singleton* sets. Typical metrics for the target vector space are either the Chebyshev metric, that measures the distance between vectors as the maximum absolute-value distance between corresponding coordinates, or any of the L_p metrics (of which the Chebychev metric is a limit).

Denote the reference points $p_1, p_2, \ldots, p_m$. Then with the Chebyshev metric we have

$$\begin{aligned} d(a_1, a_2) &\approx d'(\psi(a_1), \psi(a_2)) \\ &= \max\nolimits_{i=1,\ldots,m}\{|d(a_1, p_i) - d(a_2, p_i)|\} \end{aligned}$$

And with the Euclidean metric (L_2):

$$\begin{aligned} d(a_1, a_2) &\approx d'(\psi(a_1), \psi(a_2)) \\ &= \sqrt{\sum_{i=1,\ldots,m} (d(a_1, p_i) - d(a_2, p_i))^2} \end{aligned}$$

In practice, after choosing the m reference points (also called *pivots*), we pre-compute the $n \cdot m$ distances between each of the n services and each of the m pivots. These distances are then used to calculate all service-service similarities, which are stored in an $n \times n$ matrix as before. The new cost is now $O(m \cdot n)$. Assuming $m < n$, it is an improvement over $O(n^2)$.

7.2 Evaluation

Two aspects of the heuristic require validation. First, it must be demonstrated that the method for constructing similarity measures for domains of arbitrary complexity, described in Section 5.1, generates measures of good quality. Then, it must be demonstrated that the k-nearest neighbor method described in Section 7.1 arrives at high quality results at low computational costs. This section describes experiments that seek to validate both these aspects.

The first challenge is to design an experiment that will determine whether our similarity measures are successful in conveying the similarity *inherent* in a set of services. That is to say, the set of services has an inherent *structure*, and the issue is whether our similarity measure can detect this structure. We translated this problem to a problem of *classification*: Assume a repository signature $\mathcal{S}$ of n services, and a grouping of the services into m distinct groups, where the services in each group are constructed to be inherently similar to each other. Now, given a service $s \in \mathcal{S}$, our similarity measure was used to find the k services in $\mathcal{S}$ that are most similar to s. The set of k discovered services was compared to the group to which s belongs, using the measure of *precision*: the ratio of services that are indeed in the group of s to the total number k of services discovered.

Two independent experiments were conducted, each with a different output signature. In the first experiment the services had an output signature of the type $(S, \{R\})$; that is, the output of each service was a character string followed by a set of numbers. In the second experiment the services had an output signature of the type $(R, \{S\}, \{S\})$; that is, the output of each service was a triplet: a number followed by two set of strings (for example, a service that receives the GPS coordinates of a location and returns the Zip code of that location plus a set of movie theaters and a set of restaurants in that area).

To imbue services in the same group with similar behaviors, each group of services was randomly assigned the parameters of a distribution, and then each service in the group randomly chose values for its behavior instances from that distribution. For example, assume the output includes a set of numbers. The parameters of the distribution are two ranges: a range from which the cardinality of the set is chosen, and a range from which the values of the elements of the set are chosen. Thereafter, for each service in the group, a cardinality is randomly chosen from the first range and a corresponding set of values is randomly chosen from the second range.

The parameters used in both experiments were as follows. The set $\mathcal{S}$ consisted of n=1,000 randomly generated services, where each service was described in a behavior sample of 8 instances.[1] Then, m=10 groups of equal size were generated, with each group consisting of 100 services. The $n \times n$ similarity matrix was then computed, and a total of 278 queries were attempted (this number was chosen to assure statistical significance that is higher than 0.95). The precision of each search was calculated with neighborhood sizes of $k = 1$, 3, 5, and 10. For the embedding, we used 10 randomly chosen pivot services (this choice is justified later). The $n \times n$ matrix was constructed as described at the end of Section 7.1, and the same 278 queries were attempted and scored. To assure statistical significance, this entire experiment was repeated 100 times.

[1]As mentioned in Section 7.1, tables of this magnitude have been shown to be effective samples.

7.2.1 How Good are our Recommendations?

Figure 1 plots the results of recommending k-nearest neighbors from both the original space of services and from the target vector space. The horizontal axis is the size of k, and the vertical axis is the overall precision. Two plots reflect first experiment and two plots reflect the second experiment.

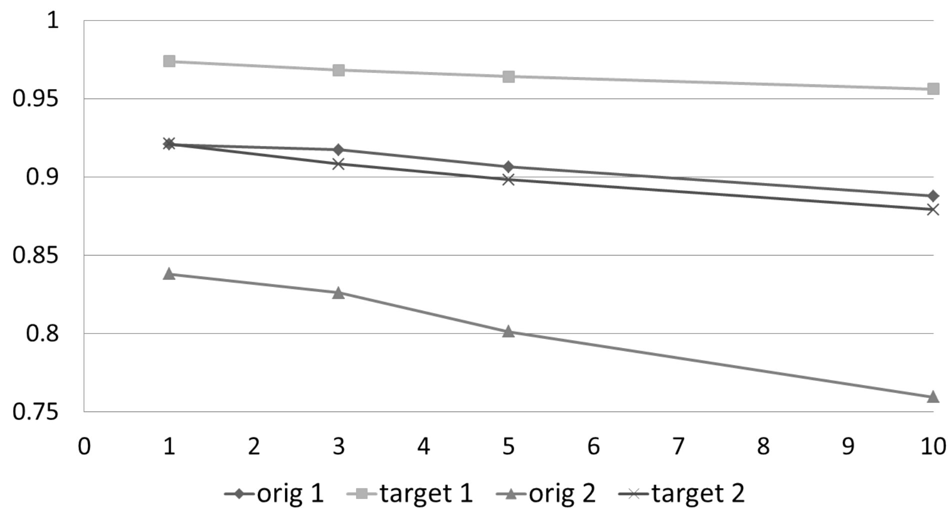


Figure 7.1: Precision at 4 neighborhood sizes

Several interesting observations are due. First, quite surprisingly, in both experiments, the performance in the target space was better than the performance in the original space. Whereas we were hoping that the embedding will improve performance (performance is discussed next) while not harming precision too much — in effect, precision improved by 5–6 percentage points. Second, across all levels of k this precision (in the target space) is kept at the impressive range of 88%–97% — a definite validation of our heuristic, given that a *random* assignment of services to groups, since there are 10 groups, would have only 10% chance of being correct. Third, performance in the second experiment was clearly

lower than in the first experiment. This can be attributed to the fact the second signature involves more string comparisons than the first signature. The services discovered and presented to the programmer bear similarity to search engine results. Good information retrieval algorithms give higher precision in earlier pages. The fourth observation is that the decline in precision with the increase in k is of a similar nature. Indeed, it is evidence that our methods percolate the better recommendations to the top.

Overall, it can be concluded safely that our methods — both the design of the signature metric and the approach to discovering the best substitute services – are effective.

7.2.2 How Much Faster is the Target Space?

The improvement in quality after embedding the space of services in a vector space was surprising. Nonetheless, recall that the main reason for this transformation was to improve performance. We expected performance to improve because (1) fewer similarities would be calculated, and (2) the calculation of each similarity would be simpler. For validation, we measured the time (minutes of CPU time) that was required for a nearest neighbor search in the original space and in the target space. This time included 10 runs, each requiring completing the $n \times n$ similarity matrix, and, for each of 278 queries, finding the 10 most similar services. We then formed this ratio:

$$Performance\ Gain = \frac{Time\ in\ original\ space}{Time\ in\ target\ space}$$

The first experiment yielded a ratio of 83.17/1.43 = 58.16 (98.3% reduction), and the second experiment yielded a ratio of 287.00/3.29 = 87.23 (98.8% reduction). Such gains in performance, coupled with the previously discussed improvement in precision, testifies to the advantage of using metric space embedding.

7.2.3 How Many Pivot Services are Needed?

As observed at the end of Section 7.1, the gain in performance achieved by the embedding method is in reducing n^2 to $n \cdot m$. It is therefore beneficial to keep the number of reference points m low. We tested 8 different numbers of pivots: 1, 5, 10, 30, 100, 200, 500 and 1,000 (recall that the total number of services is 1,000). Figure 2 plots the precision in each experiments for all but the largest 3 numbers (which were essentially the same as for 100). Precision was measured for the case $k = 1$ (i.e., search for the most similar service).

As can be observed, in both experiments, precision exceeds 0.92 when only 10 pivots are used, and it exceeds 0.96 when 30 pivots are used (it is almost flat thereafter). In other words, good results were achieved with an embedding that uses just 1–3% of the services as its reference points. This accounts for the large performance gains observed earlier. Again, results for the first experiment were slightly better.

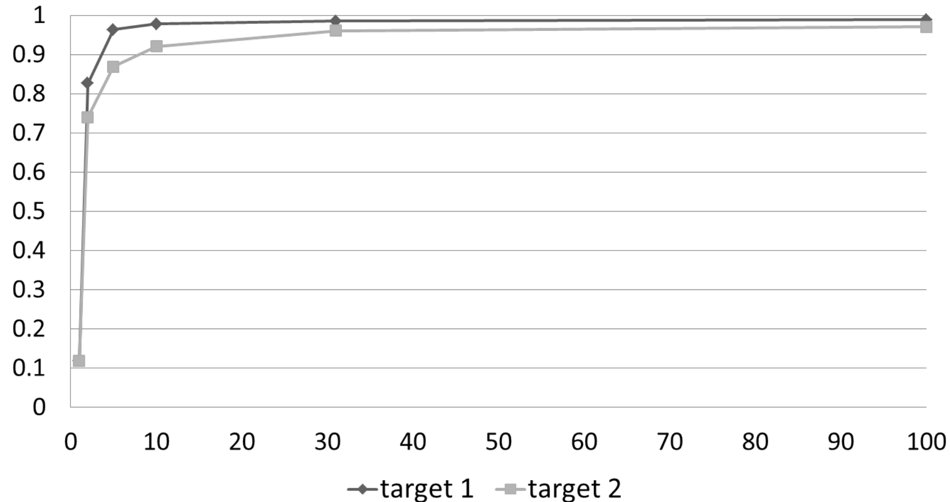


Figure 7.2: Precision at different number of pivots

7.2.4 How do Exceptions Affect Recommendation Precision?

Finally, recall that our model allows for the possibility of exceptions in service behavior. Therefore, it is important to validate the robustness of our methods in the presence of exceptions. Figure 3 plots, for each experiment, the recommendation performance in both the original space of services and the target vector space when exceptions are introduced into service behavior.

The experiment was done with a neighborhood of $k = 1$ with exception percentages of 1%, 5%, 10% and 20%. For comparison, the plots also include precision when there are no exceptions (0%). As before, performance in the target vector space was consistently higher, and performance in the first experiment was noticeably better. Observe that at exception rates under 10% precision is kept above 0.79. As one is unlikely to deploy services with exception rates higher that 10%, these results are quite encouraging.

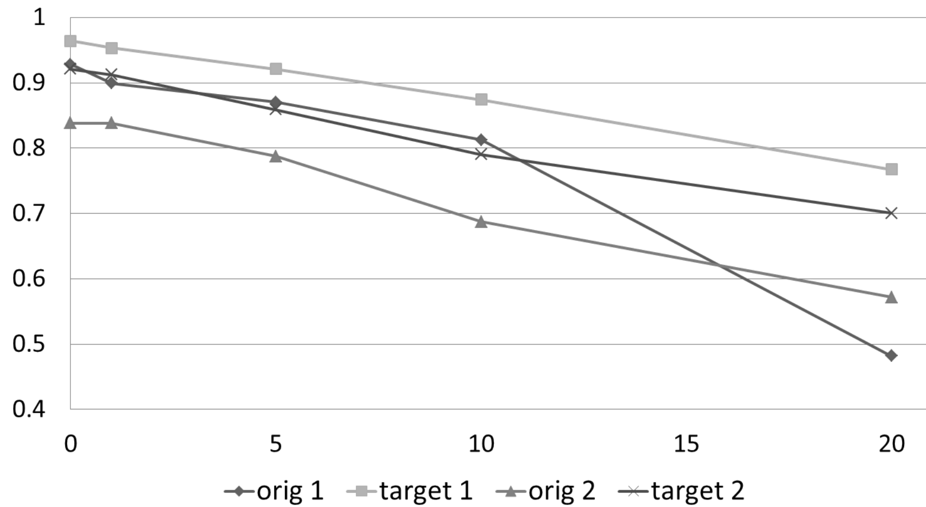


Figure 7.3: Precision at 4 percentages of service exception

Chapter 8: A Method for Recommending Substitutions in Service Composition

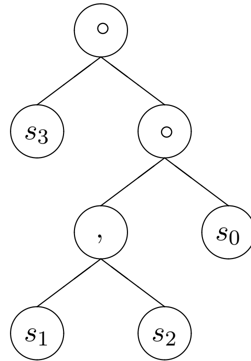
Having described a formal model for services, a definition of service similarity, and language for service compositions in chapter three, four, and five, we now turn our attention to our third issue, namely the repair of failed services in compositions. Our general premise is that a basic service has failed during the execution of a composite service, and that this service must be isolated and replaced so that the composition may be re-executed successfully. More formally, assume a composite service s has failed, and a basic service s_0 which is a component of s has been isolated as the reason for the failure. The task is to configure a new composite service s^* that will be an good substitution for s (obviously, s^* will no longer engage s_0, using other available services instead).

The main contributions of this chapter are:

- We present a method to recommend service repairs in Section 8.1.
- Then in Section 8.2, we evaluate our method on synthetic data.

8.1 Approach

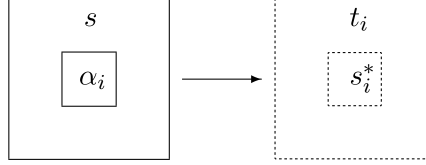
To describe our method, it is beneficial to view composite services as *trees*, in which basic basic services are leaf nodes and the *compose* and *combine* operators are internal nodes. For example, the composite service $s = s_3 \circ (s_1, s_2) \circ s_0$ is represented with the tree



When a basic service (a leaf of the tree) fails, we consider not only replacing this service, but also every sub-expression (sub-tree) that incorporates this service. This reflects the fact that there might be a service available in the repository that can substitute effectively for an entire sub-expression. As an example, assume the failed service calculates the Zip code for a given city, but, in the given composite service, this service is composed with a service that provides the temperature for a given Zip code. It might be more beneficial to substitute the composition of both with a service that provides the temperature for a given city.

8.1.1 Analyzing Repair Options

Let s be composite service that incorporates a failed basic service s_0, and let $\alpha_1, \ldots, \alpha_n$ be the sub-expressions of s that incorporate s_0. In particular, α_1 is simply the failed service s_0 and α_n is the entire composite service s. We transform s into a new composite service t_i by substituting a sub-expression α_i with a new service s_i^* which is compatible with α_i and which behaves "most similarly". Note that whereas α_i is possibly a composite service, s_i^* is a basic repository service. It should be intuitively clear that there is no advantage in considering sub-expressions that do not incorporate the failed service. We now have a set $t_1, \ldots, t_n$ of "most similar" reconfigurations of the original composite service s, and we must choose the"most similar" among them.



We use the intuitive term "most similar service"; recall in chapter five, we formalize the concept of service similarity. To summarize, consider two *compatible* services $s_1 : A \longrightarrow B$ and $s_2 : A \longrightarrow B$. Assume a metric d is defined that measures the distance between every two elements of B.[1] The similarity between two elements of B is defined as the reciprocal of the distance: $sim(b_1, b_2) = 1/d(b_1, b_2)$. The *similarity* between the two services is defined as the average similarity of their outputs for all possible inputs:

$$sim(s_1, s_2) = \frac{1}{|A|} \sum_{a \in A} sim(s_1(a), s_2(a))$$

When only *samples* of the behavior are available, the average is on the instances of the behavior samples (note that both samples should use the same values of A). For more details, refer back to chapter five and six, where this formalization is shown to reflect the inherent similarity among services.

8.1.2 Repair Procedure

Our approach to repairing failed composite services can now formalized in this procedure.

Input: A composite service s that incorporates a failed basic service s_0.

Output: A composite service t that optimally substitutes for s and does not incorporate s_0.

Step 1: Extract the sub-expressions $\alpha_1, \ldots, \alpha_n$ of s that incorporate s_0.

Step 2: For each sub-expression α_i $(i = 1, \ldots, n)$:

[1]For example if B is a domain of numbers, d could be the absolute value of their difference.

1. Construct its estimated behavior table from the behavior tables of the basic services.[2]

2. For each repository service s_i that is compatible with α_i calculate the behavior similarity $\Phi(\alpha_i, s_i)$.[3]

3. Let s_i^* denote the repository service s_i for which the behavior similarity is highest.

4. Replace the sub-expression α_i with s_i^*. Let t_i denote the result.

Step 3: For each composite service t_i $(i = 1, \ldots, n)$:

1. Construct its estimated behavior table from the behavior tables of the basic services.

2. Calculate the behavior similarity $\Phi(s, t_i)$.

Step 4: Let t denote the composite service t_i for which the behavior similarity is highest. Return t.

8.1.3 Present Recommendations to Programmer

This procedure describes how to find the best repair. Pragmatically, our method applies this procedure to derive a *ranked list* of repairs that are recommended to the programmer. This provides the programmer the opportunity to apply his judgment. In the previous example, the output might look like

1. $s_0 \longrightarrow s_{11}, s_{46}, s_{23}$

2. $(s_1, s_2) \circ s_0 \longrightarrow s_{31}, s_{18}, s_{123}$

3. $s_3 \circ (s_1, s_2) \circ s_0 \longrightarrow s_{312}, s_{12}, s_{65}$

where for each sub-expression a list of three repairs are proposed, in order of similarity.

[2] These are available from the service repository $\mathcal{R}$.

[3] These require the behavior of α_i which was estimated in the previous step, and the behavior of s_i which are available from the repository.

8.2 Evaluation

Two aspects of the repair procedure need validation so an extensive experiment was designed for each. The first aspect concerns the quality of an intermediate construct used in the procedure, and the second aspect concerns the quality of the suggested substitutions.

8.2.1 Are Compositions of Behavior Samples Good Estimations for Behavior of Service Compositions?

In Step 2.1 and in Step 3.1, the repair procedure estimates the behavior table of a composite service, from estimated behavior tables of its component services (Equations 6 and 7). How good are such estimates? Our experiment is as follows:

1. Create n basic services in each of two classes, where each service in the first class can be composed with each service in the second class; e.g., n services of the type (A, B) and n services of the type (B, C). Each of these services has a considerable number of instances; we assume the number of instances in each of the services is m.[4] All samples (estimated behaviors) were constructed with k instances each.

2. Compose each service in the first class with each service in the second class, for a total of n^2 compositions (note that each composition has m instances).

3. Sample the behavior of each of the basic $2n$ services, and each of the n^2 composite services.

4. Compose each of the n behavior samples in the first class with each of the behavior samples in the second class, for a total of n^2 samples of the compositions.

5. Compare the n^2 compositions of samples with the n^2 samples of compositions.

Clearly, a sample of the composition of two behaviors will generally be different from the composition of two samples of the same two behaviors. But we measure the quality of

[4]The two classes need not have the same number of services and the number of instances in each service need not be the same.

the method by whether the two sets of n^2 behavior samples *cluster* similarly. We partition each set into an equal number of clusters (we use $\sqrt{n}$ clusters). We now check how *similar* are the clusterings. This is done by calculating the proportion of situations in which two samples that are in the same cluster in one clustering — are also in the same cluster in the other clustering.

To test the robustness of the method, two statistical properties of the tables, called *composability* q and *range* r, were also varied. Note that the table (A, B) is always keyed on its A column, and the table (B, C) is always keyed on its B column. First, assuming the cardinality of the domain of B is p, the *composability* of the two services is defined as the ratio $q = m/p$. Composability is therefore the probability that a random instance of the first table will find a match in the second table. When $q = 1$ the second table covers the entire domain B and hence all instances in the first table find a match in the second table. Second, intuitively, clustering of the compositions (A, C) translates to clustering of sequences of length m of values from the domain of C. Such clustering is affected by the size of the domain, which we denote r. Obviously, narrow ranges create fewer clusters.

The experiment adopted initial values $n = 25$, $m = 1,000$, $k = 10$, $q = 1$ and $r = 10$; that is, we defined 25 services of type (A, B) and 25 services of type (B, C). Each service was described by a behavior table of 1,000 instances and an estimated behavior table of size 10 (1% sample). The composability was perfect and the number of different C values was set to 10.

The number of compositions was therefore $n^2 = 525$. These 525 compositions were clustered twice: by the estimates of the compositions (this could be considered the "actual") and by the compositions of the estimates (our method).

To measure the agreement between the two clusterings, we observed the $525 \cdot 525 = 275,625$ possible *pairs* of compositions; when a pair fell into the same cluster in both clusterings it was scored a *success*. Clustering agreement was then defined as the proportion of successes. This entire experiment was repeated 30 times and the agreement scores were averaged. The final score was 40%. To interpret this score, we compared the first clustering

with a random clustering. This experiment was also repeated 30 times and the agreement scores were averaged. The final score was only 22%. That is, our quick method of obtaining behavior samples of composite services performs 80% better than random.

We then repeated the experiment with different values for m, k, r and q. The results are tabulated below:

m	100	41%
	500	41%
	5,000	35%
k	2	26%
	4	32%
	8	37%
	16	46%
	32	53%
r	100	40%
	1,000	39%
	10,000	42%
q	0.5	39%
	0.2	41%
	0.1	40%
	0.05	36%

As can be observed, varying the value of m, r and q had little impact on the cluster agreement scores, implying that the method is not sensitive to the size of the behavior tables or to their statistical properties. However, the method seems to correlate positively with the size of the sample, which is rather to be expected.

8.2.2 Are Repair Recommendations Accurate?

In Steps 2.3 and in Step 4 the repair procedure finds the "best" substitution for a sub-expression. In Step 2.3 it is a repository service that substitutes a sub-expression that incorporates the failed component; In Step 4 it is a newly constructed composite service that substitutes the original composite service. Our second experiment validates that the procedure indeed suggests high quality substitutions.

In this experiment, again, we create two classes of composable services. However, the n services in each class are generated so that they fall into $\sqrt{n}$ subsets, each with $\sqrt{n}$ services, where the services in each subset *behave similarly.*[5] We then compose each service in the first class with each service in the second class for a total of n^2 compositions. Since each class has been divided into $\sqrt{n}$ subsets of similar semantics, the n^2 compositions are now divided into n subsets of n similar services.

We now choose at random a service c from the group of n^2 composite services, we compare this service with the other $n^2 - 1$ composite services, and we select the service c' with the highest behavior similarity Φ. If c' and c are in the same behavior group it is scored a *success.*

This experiment adopted the same values of $n = 25$, $m = 1,000$, $k = 10$, $q = 1$ and $r = 10$; and was repeated 30 times. The overall ratio of success was 17%; that is, in only 17% of the cases, our metric for behavior similarity selected a service of similar semantics. However, the rate of success increased to 35% when a success was defined "at least one of the 3 closest neighbors of c is in the same behavior group as c". With 5 closest neighbors it increased to 53%, and with 10 closest neighbors it increased to 85%. To summarize, when a composition of two services is to be replaced, in 85% of the time a proper substitution is included among the top 10 suggestions. Recall that there could be 524 suggestions, and a random choice of 10 services would have a chance of only about 33% to include one from the similarity group of c.

[5] The two classes need not be partitioned into the same number of subsets, and each subset need not have the same number of similar services.

Chapter 9: Conclusion

In this dissertation, we explored the problem of failure in service-oriented software. More formally, given a failed service s_0, and assuming available services are registered in a repository $\mathcal{R}$, the problem is to recommend another service s^* (or set of services) from $\mathcal{R}$ such that s^* would be a "good" substitute for s_0. To address this problem, our idea is to describe service functionality by the observable relationship between it's input and output. Our stated thesis is service behavior can be modeled, sampled economically, and compared efficiently. Then, given a service described in our model, we can recommend substitutions (found in a repository) that behave similarly.

In this final chapter, we recount issues related to the service repair problem, summarize our main contributions towards a solution, describe future research directions, and conclude with final remarks. Next we summarize the major findings that support our thesis.

1. While keeping sample size low, are samples of behavior accurate reflections of service functionality?

   Our results indicate that our approach is viable, with cluster accuracy exceeding 75% using as few as 16 sample tests points. Altogether, we show that reliable behavior can be learned from relatively small samples; often with as few as 10 to 32 behavior instances.

2. While decreasing recommendation time, are searches for service substitutions accurate?

   Using synthetic services, our experiments show the precision of discoveries to be between 88% and 97% for individual services. In addition, our experiments show that the metric space embedding method reduces search cost by as much as 98%.

3. While services are in compositions, are repair recommendations accurate?

   Our experiment showed that when the list of recommended repairs is of length 10, in 85% of tests it recommended a repair that had the same semantics as the failed composition.

9.1 Summary of Contributions

We framed our research according to three research issues: service service behavior, service substitution, and service composition. Furthermore, in the course of our investigation, we presented a model for representing information services, a definition for measuring the similarity between two services, and a language for expressing compositions of services. With these abstractions, we presented a heuristic to a representative sample of behavior while keeping sample size small; a method for recommending substitutions along with a heuristic for improving the performance of our recommendation process; and finally a method for recommending service repairs in service compositions.

The following is an enumeration of the research contributions described throughout this dissertation.

- In Section 3.1 and 3.2, we present a definition of services that models the syntax and semantics of services.
- Then we present two pragmatic adjustments to our model in Section 3.3.
- We introduce a language service compositions in Section 4.1.
- In Section 4.2, we define the mapping from service expressions to behavior tables.
- We present in Section 5.1 our method for measuring the distance between elements of domain expressions.
- Then, we describe our definition of service similarity in Section 5.2 which we handle exceptions in behavior.

- In Section 6.1, we present a method to extract a sample of service behavior by means of testing and then describe our heuristic for detecting when we have enough service instances.
- Then, we present the results of our experiments that indicate that our approach is viable, with cluster accuracy exceeding 75% using as few as 16 sample tests points in Section 6.2,.
- Additionally in Section 6.2, statistical significance tests validate that the resulting service clusters are representative of service behavior and do not occur by chance.
- Our approach to the issue of service substitutions is described in Section 7.1.
- Section 7.2 details and analyzes the experiments that validates our heuristic.
- We present a method to recommend service repairs in Section 8.1.
- Then in Section 8.2, we evaluate our method on synthetic data.

9.1.1 Summary of Issues Explored

In this next section, we provide a brief recitation of the issues explored in this dissertation and our approach toward a solution.

Service Behavior

We presented a heuristic for sampling service behavior by testing. Our solutions do not rely on the service descriptions provided by their authors and they avoid common information retrieval techniques. We conducted two tests: one with a set of weather services and another with a set of stock quoting services. Both tests validated our heuristic yielding high accuracy at surprisingly small test sizes.

Service Substitution

The increased reliance on publicly available services for creating *ad hoc*, low-cost applications brought about an urgent need for locating service substitutes. As services are usually discovered in large repositories, this problem is naturally stated in terms of a k-nearest neighbors problem, which, in turn, requires a formal method for measuring service similarity. In our opinion, it is more productive to gauge similarity on the basis of the intrinsic behavior of services than on meta-information that is associated with services.

In this dissertation, we defined a formal, yet pragmatic method of measuring similarity, and we demonstrated that services discovered on the basis of this measure are indeed, with very high level of precision (even in the presence of service exceptions). Additionally, we cast the recommendation task as a k-nearest neighbors problem then we showed how this metric space can be approximated in a vector space created by Lipschitz embeddings, with dramatic gains in performance.

Service Compositions

Lastly, we addressed the issue of substitutions in service compositions, and proposed a method for recommending service repairs. In addition, for each portion of the composition that incorporated the failed service, a list of possible substitutions was recommended in order of similarity to the original composition. With these recommendations, programmers may choose a repair that best suites their needs.

9.2 Future Directions

While we believe this dissertation is a successful step toward the repair of service-oriented software, we can envision seven future research possibilities that could improve our methods.

1. Our repair procedure performs an exhaustive search for the optimal solution. In the case of large service repositories, such a search could become expensive. So future research could explore additional heuristics to reduce the search space.

For example, the use of Lipschitz embeddings with a rather small number of pivots brought about a large gain in performance. Another research direction is to examine further methods for improving the performance of searches; for example, using *vantage-point trees* to locate the most similar services.

2. Our model defines a static view of service behavior. Since repository services could evolve there should be a mechanism that would allow periodic updates of stored behaviors. In addition, we could consider extensions to our model that could represent the temporal nature of some services.

3. During the sampling process, we could apply testing techniques to account for coverage in service functionality; that is, select input values that elicit more productive areas of service behavior.

4. We partitioned the entire repository based on the inputs and outputs of the services. We note that frequently a service that has been assigned to one partition could easily be *adapted* to a service that is comparable to services in *other* signatures. This could be done by an adaptation of its signature. The simplest adaptation is to ignore one of the service outputs. The "new" service then becomes "eligible" for membership in another signature.

5. Our methods assume a services failure; however, we could consider situations in which new services become available that outperform services currently in use. In other words, *what works today, could be made to work better tomorrow.* Thus an obvious extension of our similarity measure is to incorporate quality of service parameters, such as reliability or response time, into the definition of similarity. Then, the overall benefit of using a particular service s is indicated by a linear (weighted) combination of its quality and its behavior.

6. Our experimentation used services that were synthesized. While this allowed us to scale up recommendations to a very large number of services, it would be desirable to apply our solutions in an environment of actual services.

7. Finally, our goal is to assist programmers in maintaining the operability of service compositions. Towards this goal, it would be beneficial to conduct a user study, in which an implementation of our methodology would be available to teams of actual users.

9.3 Final Remarks

The main contributions of this dissertation are published in various software engineering research venues. The following is an enumeration of published research and the venues in which they appear.

1. 8th IEEE International Conference on Services Computing (SCC) '11 – Discovering Service Similarity by Testing [78].

2. 4th IEEE International Conference on Services Oriented Computing and Applications (SOCA) '11 – Learning Service Behavior with Progressive Testing [79]. Winner of the Best Paper award.

3. 20th IEEE International Conference on Web Services (ICWS) '12 – Efficient Service Substitutions with Behavior based metrics [80].

4. International Journal of Web Services Research (IJWSR) '13 – Efficient Measurement of Service Similarity [81].

5. 8th IEEE International Symposium on Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA) '14 – Recommending Service Repairs [82].

Our work is based on a minimalistic model, which could serve as a basis for a more elaborate system that would incorporate additional features and functionalities. Indeed, we view the problem addressed in this dissertation as part of a larger goal. Namely, how to construct a system that (1) facilitates the maintenance of service compositions, and (2) monitors their operation continuously, and (3) makes repair recommendations to programmers as needed.

The results have been encouraging and suggest that expanding the scope of this work may offer additional rewards. The model and methods that we developed can be applied to other problems in the area of service-oriented programming. For example, our methods can be adapted to provide a search tool for services based on examples. As another example, the similarity measures could be used to interconnect services in a semantic network, allowing users to *explore* large networks of services.

Bibliography

Bibliography

[1] H. Zhao and P. Doshi, "Towards automated RESTful web services composition," in *Proceedings of ICWS 2009, IEEE International Conference on Web Services.* IEEE, 2009, pp. 189–196.

[2] S. Haider, B. Ballester, D. Smedley, J. Zhang, P. Rice, and A. Kasprzyk, "Biomart central portalunified access to biological data," *Nucleic acids research*, vol. 37, no. suppl 2, pp. W23–W27, 2009.

[3] J. Zhang, R. K. Madduri, W. Tan, K. Deichl, J. Alexander, and I. T. Foster, "Toward semantics empowered biomedical web services," in *Proceedings of ICWS 2011, IEEE International Conference on Web Services.* IEEE, 2011, pp. 371–378.

[4] S. Cohen-Boulakia and U. Leser, "Search, adapt, and reuse: the future of scientific workflows," *ACM SIGMOD Record*, vol. 40, no. 2, pp. 6–16, 2011.

[5] D. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: data mashups for intranet applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 2008, pp. 1171–1182.

[6] J. Wong and J. Hong, "Making mashups with marmite: towards end-user programming for the web," in *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 2007, p. 1444.

[7] A. Riabov, E. Boillet, M. Feblowitz, Z. Liu, and A. Ranganathan, "Wishful search: interactive composition of data mashups," in *Proceeding of the 17th international conference on World Wide Web.* ACM, 2008, pp. 775–784.

[8] R. Ennals and D. Gay, "User-friendly functional programming for web mashups," in *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming.* ACM, 2007, p. 234.

[9] C. Olston, S. Chopra, and U. Srivastava, "Generating example data for dataflow programs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* ACM, 2009, pp. 245–256.

[10] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, 2005.

[11] I. Neamtiu and T. Dumitras, "Cloud software upgrades: Challenges and opportunities," in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the.* IEEE, 2011, pp. 1–10.

[12] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou, “On the evolution of services,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 3, pp. 609–628, 2012.

[13] J. Lu, Y. Yu, D. Roy, and D. Saha, “Web service composition: A reality check,” in *Web Information Systems Engineering–WISE 2007*. Springer, 2007, pp. 523–532.

[14] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “Sober: statistical model-based bug localization,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[15] H. Cleve and A. Zeller, “Locating causes of program failures,” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 342–351.

[16] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.

[17] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, “Bugfix: A learning-based tool to assist developers in fixing bugs,” in *Program Comprehension, 2009. ICPC’09. IEEE 17th International Conference on*. IEEE, 2009, pp. 70–79.

[18] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, “Applying classification techniques to remotely-collected program execution data,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 146–155, 2005.

[19] A. E. Hassan, “The road ahead for mining software repositories,” in *FoSM 2008, Frontiers of Software Maintenance*. IEEE, 2008, pp. 48–57.

[20] M. B. Blake and M. F. Nowlan, “Taming web services from the wild,” *IEEE Internet Computing*, vol. 12, no. 5, pp. 62–69, 2008.

[21] J. Sillito, G. C. Murphy, and K. De Volder, “Asking and answering questions during a programming change task,” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[22] M. Papazoglou and W. van den Heuvel, “Service oriented architectures: approaches, technologies and research issues,” *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007.

[23] C. Walton, *Agency and the semantic web*. Oxford University Press, USA, 2007.

[24] J. Andersson, R. De Lemos, S. Malek, and D. Weyns, “Modeling dimensions of self-adaptive software systems,” in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 27–47.

[25] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, “Query optimization over web services,” in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 355–366.

[26] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi, “Optimization of multi-domain queries on the web,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 562–573, 2008.

[27] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, “Software engineering for self-adaptive systems: A research roadmap,” *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

[28] D. Ardagna and B. Pernici, “Adaptive service composition in flexible processes,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 369–384, 2007.

[29] S. S. Pillai and N. C. Narendra, “Optimal replacement policy of services based on markov decision process,” in *Proceedings of SCC-2009, IEEE International Conference on Services Computing*, 2009, pp. 176–183.

[30] Q. Yu and A. Bouguettaya, “Framework for web service query algebra and optimization,” *ACM Transactions on the Web (TWEB)*, vol. 2, no. 1, p. 6, 2008.

[31] A. Pahlevan, S. Chester, A. Thomo, and H. Muller, “On supporting dynamic web service selection with histogramming,” in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*. IEEE, 2011, pp. 1–8.

[32] T. Yu, Y. Zhang, and K.-J. Lin, “Efficient algorithms for web services selection with end-to-end qos constraints,” *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 6, 2007.

[33] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa, “A framework for utility-based service oriented design in SASSY,” in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, 2010, pp. 27–36.

[34] T. Espinha, A. Zaidman, and H. Gross, “Understanding service-oriented systems using dynamic analysis,” in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*. IEEE, 2011, pp. 1–5.

[35] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 81–92.

[36] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 215–224.

[37] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, “Automatic workarounds for web applications,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 237–246.

[38] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, “A search engine for finding highly relevant applications,” in *Proceedings of ICSE 2010, ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 475–484.

[39] S. P. Reiss, “Semantics-based code search,” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 243–253.

[40] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.* ACM, 2010, pp. 157–166.

[41] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30.* VLDB Endowment, 2004, pp. 372–383.

[42] M. O'Searcoid, *Metric spaces.* Springer, 2006.

[43] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity search.* Springer, 2006.

[44] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *Proceedings of the 30th International Conference on Very Large Data Bases.* VLDB Endowment, 2004, pp. 372–383.

[45] M. B. Blake and M. F. Nowlan, "A web service recommender system using enhanced syntactical matching," in *Proceedings of ICWS 2007, IEEE International Conference on Web Services.* IEEE, 2007, pp. 575–582.

[46] F. Liu, Y. Shi, J. Yu, T. Wang, and J. Wu, "Measuring similarity of web services based on WSDL," in *Proceedings of ICWS 10, International Conference on Web Services.* IEEE, 2010, pp. 155–162.

[47] A. Günay and P. Yolum, "Structural and semantic similarity metrics for web service matchmaking," in *E-Commerce and Web Technologies, LNCS 4655.* Springer, 2007, pp. 129–138.

[48] Z. Shen and J. Su, "Web service discovery based on behavior signatures," in *Services Computing, 2005 IEEE International Conference on*, vol. 1. IEEE, 2005, pp. 279–286.

[49] M. Junghans, S. Agarwal, and R. Studer, "Behavior classes for specification and search of complex services and processes," in *Proceedings of ICWS 2012, IEEE International Conference on Web Services.* IEEE, 2012, pp. 343–350.

[50] D. Grigori, J. C. Corrales, and M. Bouzeghoub, "Behavioral matchmaking for service retrieval," in *Proceedings of ICWS 2006, IEEE International Conference on Web Services.* IEEE, 2006, pp. 145–152.

[51] Q. Yu and M. Rege, "A relational approach for efficient service selection," in *Proceedings of ICWS 2009, IEEE International Conference on Web Services.* IEEE, 2009, pp. 719–726.

[52] A. Tversky, "Features of similarity," *Psychological review*, vol. 84, no. 4, p. 327, 1977.

[53] R. N. Shepard, "Toward a universal law of generalization for psychological science," *Science*, vol. 237, no. 4820, pp. 1317–1323, 1987.

[54] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM, 2009, pp. 213–222.

[55] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 111–120.

[56] G. R. Hjaltason and H. Samet, "Properties of embedding methods for similarity searching in metric spaces," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 5, pp. 530–549, 2003.

[57] L. Tang and M. Crovella, "Virtual landmarks for the internet," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement.* ACM, 2003, pp. 143–152.

[58] S. Kang, D. Kim, Y. Lee, S. J. Hyun, D. Lee, and B. Lee, "A semantic service discovery network for large-scale ubiquitous computing environments," *ETRI journal*, vol. 29, no. 5, pp. 545–558, 2007.

[59] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge University Press, 2008.

[60] S. R. Tilley, X. Bai, and G. A. Lewis, "Proceedings of soat-2009, first international workshop on service-oriented architecture testing," in *Proceedings of ICSM-2009, IEEE International Conference on Software Maintenance*, 2009, pp. 583–584.

[61] J. Offutt and W. Xu, "Generating test cases for web services using data perturbation," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1–10, 2004.

[62] C. Romesburg, *Cluster analysis for researchers.* Lulu. com, 2004.

[63] P. Tan, M. Steinbach, V. Kumar *et al.*, *Introduction to data mining.* Pearson Addison Wesley Boston, 2006.

[64] W. Liu and W. W.W̃ong, "Discovering homogenous service communities through web service clustering," *Service-Oriented Computing: Agents, Semantics, and Engineering*, pp. 69–82, 2008.

[65] M. Sellami, W. Gaaloul, and S. Tata, "Functionality-driven clustering of web service registries," in *Proceedings of SCC-2010, IEEE International Conference on Services Computing*, 2010, pp. 631–634.

[66] Q. Yu and M. Rege, "On service community learning: A co-clustering approach," in *Proceedings of ICWS-2010, 8th IEEE International Conference on Web Services*, 2010, pp. 283–290.

[67] Y. Park, W. Jung, B. Lee, and C. Wu, "Automatic discovery of web services based on dynamic black-box testing," in *Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference*, 2009, pp. 107–114.

[68] G. R. Santhanam, S. Basu, and V. Honavar, "Web service substitution based on preferences over non-functional attributes," in *Proceedings of SCC-2009, IEEE International Conference on Services Computing*, 2009, pp. 210–217.

[69] M. D. Ernst, R. Lencevicius, and J. H. Perkins, "Detection of web service substitutability and composability," in *Proceedings of WS-MaTe-2006, International Workshop on Web Services–Modeling and Testing*, 2006, pp. 123–125.

[70] A. Flores and M. Polo, "Testing-based process for evaluating component replaceability," *Electronic Notes in Theoretical Computer Science*, vol. 236, pp. 101–115, 2009.

[71] A. Motro, *Management of Uncertainty in Database Systems.* Addison-Wesley/ACM Press, 1994, pp. 457–476.

[72] D. B. Rubin, *Multiple Imputation for Nonresponse in Surveys.* Wiley, 2004.

[73] F. Provost, D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1999, pp. 23–32.

[74] G. H. John and P. Langley, "Static versus dynamic sampling for data mining," in *Proceedings of Second ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 367–370.

[75] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical Society*, vol. 66, no. 336, pp. 846–850, 1971.

[76] P. N. Tan, M. Steinbach, V. Kumar *et al.*, *Introduction to Data Mining.* Pearson/Addison Wesley, 2006.

[77] H. Samet, *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.

[78] J. Church and A. Motro, "Discovering service similarity by testing," in *8th IEEE International Conference on Services Computing (SCC).* IEEE Computer Society, 2011.

[79] ——, "Learning service behavior with progressive testing," in *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA).* IEEE Computer Society, 2011, pp. 1–8.

[80] ——, "Efficient service substitutions with behavior based metrics," in *20th IEEE International Conference on Web Services (ICWS).* IEEE Computer Society, 2013.

[81] ——, "Efficient measurement of service similarity," *International Journal of Web Services Research (IJWSR)*, vol. 10, no. 2, pp. 23–40, 2013.

[82] ——, "Recommending service repairs," in *2014 IEEE 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA).* IEEE Computer Society, 2014.

Curriculum Vitae

Joshua Church received his B.S. in Computer Information Systems from Bradley University in 2003. Later in 2007, he graduated with a M.S. in Software Engineering from George Mason University. His research interests include applying information retrieval techniques to software engineering problems. He has published multiple articles in this area to include willing the Best Paper award at the 4th IEEE International Conference on Services Oriented Computing and Applications (SOCA).