Discovering Context Dependent Service Models for Stateful Service Virtualization

Md. Arafat Hossain

February 2020

Swinburne University of Technology

Doctoral Thesis
Melbourne, Australia

Abstract

Today's enterprise software systems consist of many inter-connected software components and services, to support the enterprises' essential business operations.The testing of a particular component service (for its initial introduction or upgrade) requires access to other component services that the component under test depends on, to ensure the entire system works properly. Due to difficulties in achieving such access (such as availability and cost), traditional testing methods often use stubs, mock objects and/or virtual machines to achieve a limited level of access, sacrificing accuracy and scalability. In recent years, service virtualization has become an alternative and more effective solution to provide a more realistic environment for comprehensive testing.

Service virtualization enables the testing of a component service in a highly interconnected enterprise system by using "virtual" services, i.e., executable models that emulate the dependent services in a realistic manner. A number of techniques have been proposed for creating such executable service models automatically by analyzing the interaction traces between the dependent services and other components of the system. These existing techniques have been proposed for stateless services, and the creation of service models for stateful services is still unexplored, severely limiting the applicability of the service virtualization approach.

The main objective of this thesis is to develop techniques for deriving executable models for stateful services from their interaction traces by extracting and using the relevant contextual and dependency information in the interaction messages. To be able to identify the message fields and their relationships, we first need to identify the types and formats of the various messages in the service interaction traces. As such, our approach focuses on three key research issues: 1) message type identification and format extraction for request messages; 2) message type identification and format extraction for response messages, which have different challenges from request messages; 3) inference of service behavior to capture the message information and relationships necessary to generate appropriate responses for service requests that are dependent on the service state at the time.

Our first research issue concerns the identification of the types and formats of request messages. By aligning the stable message fields across the request messages and analyzing the fields' occurrence rates, we are able to identify the type field of the request messages, and classify them into type-specific clusters. Then, the format for each type of messages is extracted by identifying the recurring (key)word pattern across all messages in each of the message clusters.

The second research issue focuses on identifying the types and formats of response messages. Different from request messages, response

messages often do not have a distinctive type field, and there may be different types of response messages for a given request message. As such, a different clustering technique is developed to classify the response messages into type-specific clusters by identifying the co-occurrence of common message field patterns across the messages. Then, the format for each type or cluster of response messages is extracted by identifying the common (key)words over the messages in the cluster, excluding their repetitions in individual messages so as to accommodate the fact that response messages often contain repetitive payload entries.

After identifying the types and formats of request and response messages, the third research issue concerns the discovery of the behaviour models for stateful information services, and their use in generating response messages for any request messages. It first extracts a general request-response message protocol model (in the form of a finite state machine) by identifying and merging (generalizing) information record-specific message sequences from the given message traces. Then, it identifies the general relationships between/across message payload fields by analyzing their recurrences in the message traces. Finally, the general protocol model and payload relationships are used to instantiate a specific service model for each information record involved at runtime, to generate response messages (with appropriate formats and payload fields) for any incoming request messages concerning that record.

Our approach and techniques are evaluated on message traces collected from a range of real-world software services and systems. The evaluation results have shown that our approach has achieved significant improvements in both effectiveness (accuracy) and efficiency (performance) over existing techniques. This covers the identification of message types and formats, derivation of the service behaviour model (behavioural dependencies between messages and between message payload fields), and the generation of response messages using the behaviour model, achieving a level of realistic emulation of stateful service behaviour not seen to date.

Acknowledgements

First and foremost, all praises to the almighty Allah, for his countless blessings throughout my study.

I would like to express my sincere appreciation to my principal supervisor Prof. Jun Han, for his constant guidance and encouragement, without which this work would not have been possible. I hope, and am excited to continue my research career under his guidance in the future. I am also extremely grateful to my associate supervisors Prof. Jean-Guy Schneider, Dr. Steve Versteeg and Dr. Ashad Kabir for their valuable feedback and advice on my research activities.

My thanks also go to the members of our research group: Prof. Chengfei Liu, Prof. Christopher Leckie and Dr. Jiaojiao Jiang for their valuable suggestions and continuous support throughout my PhD study.

I would like to thank the members of the progress review committee: Prof. Yun Yang, Dr. Alan Colman and Dr. Karola von Baggo for their constructive feedback, which helped me to improve my research significantly.

I am extremely grateful to my parents for their support, love, and sacrifices to prepare me for the future. Last but not least, I thank my wife for her love, understanding, encouragement and continuing support to complete this research work.

Declaration

This is to certify that,

- This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the examinable outcome;
- To the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of the thesis;
- Where the work is based on joint research or publications, discloses the relative contributions of the respective workers or authors.

Md. Arafat Hossain
February 2020

List of Publications

Part of this dissertation has been published in the following peer-reviewed conference:

1. **Md. Arafat Hossain**, Steve Versteeg, Jun Han, Muhammad Ashad Kabir, Jiaojiao Jiang, Jean-Guy Schneider, "Mining accurate message formats for service APIs", *in Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 20-23 March 2018, Campobasso, Italy.

   – Presents the format inference approach by identifying the message type field described in Chapter 4.

Additionally, my plan is to submit the following journal papers based on the contributions made by this dissertation:

2. **Md. Arafat Hossain**, Jun Han, Jean-Guy Schneider, Muhammad Ashad Kabir, Steve Versteeg, Jiaojiao Jiang, "Extracting fine-grained formats of service messages", *ACM Transactions on Internet Technology.*

   – Presents the technique of inferring formats of the response messages by clustering the response messages of each request-based cluster and generalizing the repetitive sequence of message fields described in Chapter 5.

3. **Md. Arafat Hossain**, Jun Han, Steve Versteeg, Jean-Guy Schneider, Muhammad Ashad Kabir, "Mining service behavior for stateful service emulation", *IEEE Transactions on Software Engineering.*

   – Presents the response generation technique by inferring the behavior of the service described in Chapter 6.

Furthermore, I have contributed to the following papers during my PhD candidature:

4. Jiaojiao Jiang, Steve Versteeg, Jun Han, **Md. Arafat Hossain**, Jean-Guy Schneider, Christopher Leckie, Zeinab Farahmandpour, "P-Gram: Positional N-Gram for the Clustering of Machine-Generated Messages", *IEEE Access*, Vol. 7, June 2019, pp. 88504-88516.

5. Jiaojiao Jiang, Steve Versteeg, Jun Han, **Md. Arafat Hossain**, Jean-Guy Schneider, "A positional keyword-based approach to inferring fine-grained message formats", *Future Generation Computer Systems*, Vol. 102, January 2020, pp. 369-381.

6. Jiaojiao Jiang, Jean-Guy Schneider, Steve Versteeg, Jun Han, **Md. Arafat Hossain**, Chengfei Liu, "R-gram: Inferring Message Formats of Service Protocols with Relative Positional N-grams", *Journal of Network and Computer Applications* (In Review).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With today's technology, enterprise software systems are highly connected with many heterogeneous software components and services to support complex business processes and to meet the customers' growing requirements. Such an enterprise system needs to be tested before deployment to ensure its quality. The testing of such interconnected systems requires access to the connected components or services. But, access to these dependent services at the time of system testing is limited or even impossible due to the cost, availability or security concerns [2]. Several techniques have been proposed to facilitate the testing of such enterprise systems by simulating or mocking the behavior of the connected services. Stubs are generated to perform some basic functionality of the remote systems [3], while mock objects [4] and fake [5] provide support for testing by mimicking the real objects. Furthermore, virtual machines [6, 7] are used to provide runtime environments by replicating the physical devices for deploying the dependent services. However, these traditional techniques are unable to provide a scalable and realistic testing environment for enterprise systems [8]. In contrast, service virtualization simulates the behavior of the dependent services and provide an alternative approach to providing proper testing environments for the system under test (SUT).

Service virtualization involves the creation and deployment of "service models" that emulate the specific behavior of the dependent services, which is required to exercise the SUT. A service model stands in for a dependent service by listening to requests from the SUT and returning appropriate responses for the incoming requests with appropriate performance. Several techniques [3, 2, 9, 10, 11] have been proposed to create such service models from the actual services interaction traces that can mimic the behavior of the services by synthesizing responses for

the incoming requests. But, these techniques have only considered the creation of service models for stateless services and the creation of service models for stateful services is still unexplored.

A novel technique is presented in this thesis to creating service models for stateful services automatically by analyzing the interaction traces of the dependent services. In particular, we focus on identifying the dependency relationships between messages to determine the types of responses accurately for the incoming requests and dependency relationships between message fields to determine the proper payloads of the response messages.

This chapter is organized as follows. Section 1.1 presents a brief introduction of service virtualization. In Section 1.2, we present the characteristics of software services. Section 1.3 presents the research problem while Section 1.4 outlines the key research questions. We present the major contributions of this research in Section 1.5 and finally, Section 1.6 presents the organization of this thesis.

1.1 Service Virtualization and Its Importance

Service-oriented architecture allows the developer to use third-party services or components in developing an enterprise system [12]. But, such integration of third-party services makes the testing of the entire system more difficult as the testing requires access to the connected services, which is not always possible [2]. Moreover, the functionality of an entire system needs to be verified when any of its connected services are upgraded. Such an upgrade of the dependent services may have a cascading effect on the entire system, *i.e.*, a fault in one dependent service may trigger the failure of the entire system. For example, an unexpected inter-dependency between systems triggered a failure to the tunnel safety system when CityLink (one of the toll road operators in Melbourne) attempted to upgrade their billing system, which leads to the tunnel closure and traffic chaos [13]. Therefore, it is important that an enterprise systems' interaction with its interconnected services needs to be tested prior to deployment.

Service virtualization is a method of emulating the specific behavior of the dependent components or services in testing heterogeneous software systems. In a service virtualization environment, DevOps teams do not need to use production/actual dependent services, rather, they use the emulation or virtual services, which enables frequent and comprehensive testing of a SUT. The behavior/characteristics of a virtual service depends on the dependent/connected services of an enterprise system and it emulates the specific behavior of the dependent service, which is required to

execute the development and testing tasks. For example, virtualizing a web service requires listening for a request message over HTTP, JMS, or MQ and then returning a response message. Virtualizing a database application means that the service model is able to parse an SQL query and then return the data source rows according to the query request as the response message.

1.2 Characteristics of Software Services

A software service is a software program that runs on a server, listening for requests and returning responses based on the incoming requests. A service usually supports different types of request messages and generates different types of responses for each request message. The service requester (*i.e.*, SUT in a service virtualization environment) sends a request message and the service (*i.e.*, the service model in the service virtualization environment) is responsible for generating an appropriate response for the incoming request message. This suggests that the request message from the service requester has to follow formats defined in the service description. Thus, the service provider can identify the request type and data values from the request messages, and generate an appropriate response for the request message. Similarly, the generated responses by the service must follow formats defined in the service description, so that the receiver can parse the response messages.

A service can be classified as stateless or stateful.

Stateless Service: The result or response for a request does not depend upon the requests/operations that have been executed previously [14]. For example, the response messages (http response) of an online newspaper do not depend on the preceding requests and a reader can send http requests to connect with the server in a stateless manner.

Stateful Service: Unlike stateless service, the result or response for a request in a stateful service depends upon the requests/operations that have been executed previously [14]. An example of such services is a Banking service. An account has to be opened before getting any financial services, the response of a withdraw request depends on the requested withdrawal amount and the current balance of that account, every change made to the account is considered as a change of state for that account. Therefore, the response of a request in such a banking service depends on the preceding requests.

However, we consider the statefulness of a service as being reflected in the captured interaction trace of the service, *i.e.*, a service is considered as stateful only if the change of the service state is observable in the trace. We do not explicitly

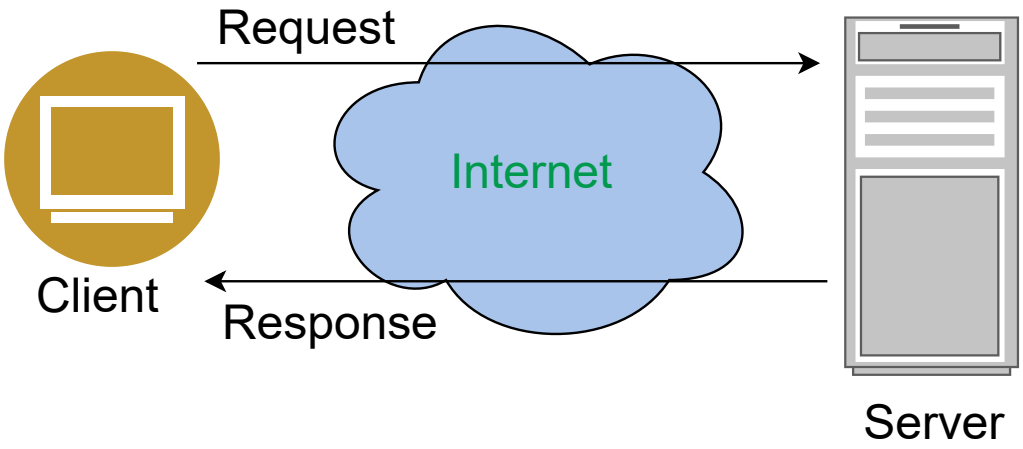


Figure 1.1: Web Service Architecture

consider the complex interplay between dependent services and the change of the service state due to such unseen interactions and only consider those effect reflected in the interaction trace. For example, a bank usually pays interest to the savings account at the end of a month and the balance is credited. But, such changes in the account are not observable in the trace as the trace is captured by intercepting the communication between a client (*i.e.*, SUT) and the service (banking service). Similarly, the financial regulatory authority may freeze an account due to suspicious activities and again, such changes are not observable in the trace.

Figure 1.1 shows a simplistic overview of how a client communicates with a web service. A client (a user or an application) usually invokes a series of request messages to the server (where the service is hosted) and gets the response message from the server. The messages (request and response) that are transmitted during the communication must follow the format as defined in the service description language. The server returns appropriate response messages to the client after identifying the type of requests from the request messages as the service commonly supports different types of request messages. Moreover, the response messages generated by the server depend on the sequence of preceding messages when the service is stateful.

1.3 Research Problem

A service model can be created manually by defining the behavior of a dependent service [3]. But, this way of defining the service model to represent the behavior of a dependent service is a tedious, time-consuming task and requires an expert with detailed knowledge of the service [11]. Also, the characteristics of the service model must be different for each different dependent service, *i.e.*, the service model needs to be defined separately for each dependent service. To construct a proper

testing environment, require the connection of many such service models one for each dependent service.

The service model can be created automatically by recording live interactions of the dependent service and then deducing the behavior of the dependent service from the recorded service interaction trace. The SUT communicates with the derived service models in the service virtualization environment instead of accessing the actual services. As such, the functionality and performance of the derived service models should reflect the actual dependent services. Therefore, the creation of the service models requires analysis and extraction of relevant information from the service interaction trace.

A number of techniques [2, 9, 10, 11] have been proposed to automatically derive service models by analyzing the service interaction trace. These techniques utilize sequence alignment algorithms [15] to find an interaction similar to the incoming request in the service interaction trace. Then, the corresponding response is transformed using similar payloads in the synthesized response message. But, these techniques do not consider the service state in synthesizing responses and hence, are unable to derive service models accurately for stateful services. As described in Section 1.2, the responses of a stateful service depend upon the preceding requests. Therefore, the creation of service models for stateful services requires the identification of the *dependency relationships among messages* to represent the behavior of the dependent services accurately. It implicitly requires the identification of the different *types of requests* and the different *types of responses* that are generated for the requests. Moreover, the message fields of the response message may correlate with the corresponding request message or the preceding requests. Thus, the service models for stateful services have to identify such *correlations between message fields* to synthesize responses with appropriate data values. However, the existing techniques for identifying symmetric fields[1] between request and response messages do not consider the message structure, resulting in invalid responses. Therefore, the identification of such correlations between message fields requires to extract *message structure* from the request and response messages.

1.4 Research Questions

In order to create service models for stateful services from the service interaction traces to provide a realistic testing environment, the following research questions

[1]The message fields containing the same payload in both the request and corresponding response message.

need to be addressed:

1. "*Can the service models identify the types and formats of the request messages?*"

   A service usually supports different types of requests and different types of responses are generated for each of them. Thus, to generate appropriate responses for the incoming requests, the derived service models have to identify the request type from the messages. Moreover, each type of request message has its own format (*i.e.*, message structure). It is also necessary to extract the formats of the request messages in order to separate the data values from the structure and consequently, identify the relationships between message fields. This question is addressed in Chapter 4 of this thesis.

2. "*Can the service models identify the different types of responses that are generated for the same request and extract the formats of the response messages?*"

   A stateful service usually generates different types of responses for the same type of request messages at different times due to the service state. Unlike the request messages, response messages do not always contain message type field. The different types of response messages have their own format as in the request messages. Moreover, the response messages usually contain *repetitive* patterns of the message fields. Therefore, the service models have to identify the different types of responses and infer formats of the response messages that are generated for the same request. This question is addressed in Chapter 5 of this thesis.

3. "*Can the service models identify the dependency between messages and between message fields for tracking the service state to synthesize responses for stateful services?*"

   In a stateful service, the responses depend upon the preceding sequence of requests in addition to the incoming request messages. It implies that a request or a sequence of requests changes the service state and thus, the service generates different responses for the same type of requests at different times. Therefore, it is required for the derived service models to identify such dependency among messages, to synthesize the responses more accurately for stateful services. Moreover, the message fields of the generated responses may correlate with the corresponding request and/or preceding messages. Thus, the service model needs to identify such correlations among message fields for

synthesizing the responses with appropriate values in the message fields. This question is addressed in Chapter 6 of this thesis.

1.5 Research Contributions

The overall contribution of this thesis the derivation of service models for stateful services, enabling the creation of a realistic testing environment for the system under test. Our approach derives service models by analyzing and extracting the dependency relationships between messages and between message fields from the service interaction trace to synthesize responses by considering the service state. More specifically, this thesis makes the following contributions:

- *A technique for inferring formats of the request messages.* It identifies the request type from the messages by analyzing the fixed and variable portions of request messages and messages are grouped into type based clusters. Then, the format for each type of request messages is extracted by identifying the patterns of message fields of all messages in the request-type based cluster.

- *A technique for extracting formats of the response messages.* Unlike the request messages, response messages do not always contain a message field to indicate the type of responses and can not use the technique for extracting formats of the request messages. As such, the responses are clustered by considering the co-occurrences of the fixed portions of the messages. Such clustering separates different types of responses that are generated for the same request. Then, the format of the responses for each cluster is extracted by identifying the common portions across the response messages, and considering the *repetitions* in each message as the response messages usually contain the repetition sequences of (key)word and data values. Such a repetition of message structure is effectively handled in format inference so that the inferred format can accept valid yet unseen messages.

- *A technique to identifying the control and data dependencies between messages and message fields respectively.* To capture the effect of service state, we infer the control dependency between messages by analyzing and extracting the contextual information (dependency) from the service interaction trace. The inferred control dependency model is used to keep track of the service state and determine the proper response type for the incoming request message. Moreover, the data values that are present in the response messages commonly have

correlations with the corresponding request message or a sequence of preceding messages (*i.e.*, data dependency). Such dependency between message fields is extracted by analyzing the data values of the request and response messages, and utilized in synthesizing responses so that the synthesized responses contain appropriate data values in the message fields.

We evaluate our approach (format inference) by applying it to a number of traces collected from both stateful and stateless service. The results show that a greater improvement in format inference over state of the art approaches. The experimental results also show that our approach (response generation) achieves significant improvement in both accuracy and efficiency over existing service virtualization techniques in synthesizing responses.

1.6 Thesis Structure

The rest of the thesis is organized as follows:

Chapter 2 motivates this research by analyzing an example trace collected from a real-world service with the underlying protocol being the Light-weight Directory Access Protocol (LDAP). The research problem is analyzed from different aspects of our thesis requirements.

In Chapter 3, we review the existing research efforts related to service virtualization. We also review the existing research efforts on extracting message structures and message dependency (*i.e.*, control dependency and data dependency) as such research works are related to virtualizing stateful services.

Chapter 4 presents our approach to inferring the format of the request messages, where message type is identified using entropy analysis and message format is extracted for each request-type based message cluster after extracting keywords from the messages of that cluster.

In Chapter 5, we present our approach to inferring the format of the response messages after clustering the responses based on the co-occurrences of message keywords.

In Chapter 6, we present our approach to inferring service behavior in order to track the state of the service. We also present an approach to utilizing the inferred service behavior model (control and data relationships) in formulating responses by instantiating the behavioral model separately for each of the *data record* maintained by the service.

Finally, in Chapter 7, we conclude this dissertation by summarizing the key contributions and discuss the directions of future research.

Chapter 2

Motivation and Problem Analysis

In this chapter, we motivate our research using a real-world example of a testing environment for an enterprise system. The testing scenario is presented in both a traditional and a service virtualization environment. We identify the key requirements for a service virtualization environment and present some examples, which will be used in later chapters of this thesis.

This chapter is organized as follows. Section 2.1 first presents an overview of interconnected software systems and then presents an enterprise system in a traditional testing environment and in a service virtualization environment. Section 2.2 presents an example interaction trace and defines the key concepts and terms that are used to describe the issues, requirements and techniques of stateful service virtualization in this thesis. Section 2.3 presents some key issues of providing virtualization environment for stateful services by analyzing the testing scenario. Based on this analysis, Section 2.4 identifies the general requirements of virtualizing stateful services. This chapter is summarized in Section 2.5.

2.1 Enterprise Software Systems and Their Testing

Software systems have grown increasingly large through connecting with many other heterogeneous services to provide support for complex business processes. For example, making a phone call requires to operate several heterogeneous systems including operational support systems (OSS) and business support systems (BSS). Several heterogeneous systems are included in the OSS, including network configu-

ration and fault management, customer activation, network security, etc. A few of the interconnected systems in the BSS are billing systems for customers, managing offers, cross-carrier transactions, etc. Moreover, a proper integration between BSS and OSS is required to provide the telecommunication services to the customers. Component-Based Software Engineering (CBSE) is an approach to develop such large and complicated systems by connecting the available and re-usable systems or components [16]. Due to the increasing popularity of the service-oriented architecture (SOA), third-party services are used as components in developing such an enterprise system [12]. The availability of third-party services provides flexibility and enables the developers to build a large and complex system. One example of using third-party services is to adopt a third-party *payment* service for an e-commerce website.

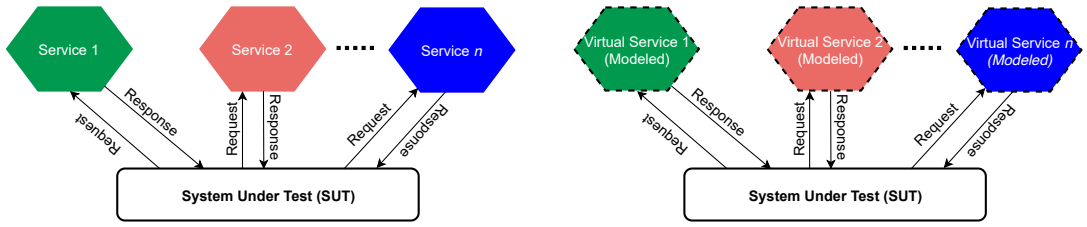
Software testing plays an important role to verify the correctness of software and confirms the accurate implementations of the requirements [17]. Every software system has to be tested and requires a confirmation from the testing team before being deployed to ensure that the system performs as expected and does not produce any unintended result [17]. Testing of an enterprise system connected with heterogeneous services requires access to those interconnected or dependent services to confirm that the entire system works properly and is ready to be released. But, the use of third-party services or components makes the testing of the entire system difficult and costly as it requires access to the connected services or components.

In a traditional testing environment, the system under test (SUT) requires access to the connected services to begin the testing (functional, integration, performance, etc.) and the software testing team has to wait for nearly completed software [16]. But, due to the need for rapid delivery, it is often impossible to wait for so long before testing. DevOps aims to shorten the software development life cycle through continuous delivery where repeated testing in a short period is required [18]. Moreover, DevOps focuses on continuous integration and delivery, which requires frequent access to the dependent services. Service virtualization offers an alternative approach to testing such complicated software systems, where "virtual" replica of the dependent services are used instead of accessing the actual services themselves.

2.1.1 Software Testing Environment

Figure 2.1 shows the communication between the system under test and services in both a traditional environment (Figure 2.1a) and a service virtualization environment (Figure 2.1b). As the Figure 2.1a shows, an enterprise system is connected

with n different services and it consumes those dependent services in a traditional testing environment. The traditional testing environment requires access to the dependent services or components to run a proper testing and hence, ensuring its quality. DevOps practices bring faster and frequent releases, which requires repeated testing and consequently, requires access to the dependent services more frequently. But, such access to the dependent services is costly and often not possible due to availability or security concerns [19]. In a service virtualization environment, on the other hand, the SUT sends request messages to the "virtual" services and gets the response messages back from these virtual services instead of getting the responses from the actual services. As Figure 2.1b shows, the SUT communicates with the virtual services during testing and the virtual services emulate the specific behavior of the corresponding dependent services to enable the testing without accessing the actual services. The "virtual (model)" services stand in for actual dependent services by listening for requests and returning the appropriate responses to facilitate the proper testing for the SUT in the service virtualization environment.



(a) Traditional testing environment **(b)** Service virtualization environment

Figure 2.1: System Under Test in Traditional and Service Virtualization Environment

2.2 Example Interaction Trace and Terminology

This section presents an example interaction trace and defines the key concepts and terminology that are used to describe the issues, requirements and techniques of service virtualization in this thesis.

2.2.1 Example Interaction Trace

Table 2.1 shows an illustrative LDAP interaction trace, which is collected from the CA Identity Manager (IM) service [1] through intercepting the communication

between a client and the IM service. The CA IM service uses the Light Weight Directory Access Protocol (LDAP) [20] to manage the digital identities of the personnel of a large organization. The client send a request message (*e.g.*, **`search`**, **`delete`**) to the server (*i.e.*, IM) and then, the server returns a response message to the client after processing the request. As Table 2.1 shows, the **`Bind`** request (*i.e.*, the first request) is used to authenticate a client, and the **`Unbind`** request (*i.e.*, the last request) is for closing the connection, which does not require a response. The request message in-between the **`bind`** and **`unbind`** requests, *i.e.*, **`search, add,`** and **`delete`** requests are used for searching an entry, adding a new entry and deleting an existing entry respectively from the IM service.

Table 2.1: Ten Interactions of a LDAP Communication Session

No.		Interactions
1	Req	LDAP **Bind** Request Message ID: 1 LDAP **Bind** Request Protocol Op LDAP Version: 3 Bind dn: cn=admin,dc=ca,dc=com Authentication Data: Authentication Type: Simple Bind Password: 1228013670
	Resp	LDAP **Bind** Response Message ID: 1 LDAP **Bind** Response Protocol Op Result Code: 0 (Success)
2	Req	LDAP **Search** Request Message ID: 2 LDAP **Search** Request Protocol Op Base DN: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com Scope: 0 (baseObject) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Done Message ID: 2 LDAP **Search** Result Done Protocol Op Result Code: 32 (No Such Object) Matched DN: ou=Finance,ou=Corporate,dc=ca,dc=com
3	Req	LDAP **Add** Request Message ID: 3 LDAP **Add** Request Protocol Op dn: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Dominic.MAJOR@ca.com mobile: 6017515 description: Customer Service objectClass: inetOrgPerson title: Administrative Operator sn: MAJOR cn: Dominic MAJOR
	Resp	LDAP **Add** Response Message ID: 3 LDAP **Add** Response Protocol Op Result Code: 0 (Success)
4	Req	LDAP **Search** Request Message ID: 4 LDAP **Search** Request Protocol Op Base DN: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com Scope: 0 (baseObject) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:

Table 2.1 Continued: Ten interactions of a LDAP communication session

No.		Interactions
	Resp	LDAP **Search** Result Entry Message ID: 4 LDAP **Search** Result Entry Protocol Op dn: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Dominic.MAJOR@ca.com mobile: 6017515 description: Customer Service objectClass: inetOrgPerson title: Administrative Operator sn: MAJOR LDAP Search Result Done Message ID: 4 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
5	Req	LDAP **Add** Request Message ID: 5 LDAP **Add** Request Protocol Op dn: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Dominic.MAJOR@ca.com mobile: 6017515 description: Customer Service objectClass: inetOrgPerson title: Administrative Operator sn: MAJOR cn: Dominic MAJOR
	Resp	LDAP **Add** Response Message ID: 5 LDAP **Add** Response Protocol Op Result Code: 68 (Entry Already Exists)
6	Req	LDAP **Add** Request Message ID: 6 LDAP **Add** Request Protocol Op dn: cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Susana.LOW@ca.com mobile: 6726920 description: Customer Service objectClass: inetOrgPerson title: Applications Administrator sn: LOW cn: Susana LOW
	Resp	LDAP **Add** Response Message ID: 6 LDAP **Add** Response Protocol Op Result Code: 0 (Success)
7	Req	LDAP **Search** Request Message ID: 7 LDAP **Search** Request Protocol Op Base DN: ou=Finance,ou=Corporate,dc=ca,dc=com Scope: 1 (singleLevel) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:

Table 2.1 Continued: Ten interactions of a LDAP communication session

No.		Interactions
	Resp	LDAP **Search** Result Entry Message ID: 7 LDAP **Search** Result Entry Protocol Op dn: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Dominic.MAJOR@ca.com mobile: 6017515 description: Customer Service objectClass: inetOrgPerson title: Administrative Operator sn: MAJOR LDAP **Search** Result Entry Message ID: 7 LDAP **Search** Result Entry Protocol Op dn: cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Susana.LOW@ca.com mobile: 6726920 description: Customer Service objectClass: inetOrgPerson title: Applications Administrator sn: LOW Search Result Done Message ID: 7 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
8	Req	LDAP **Delete** Request Message ID: 8 LDAP **Delete** Request Protocol Op dn: cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com
	Resp	LDAP **Delete** Response Message ID: 8 LDAP **Delete** Response Protocol Op Result Code: 0 (Success)
9	Req	LDAP **Search** Request Message ID: 9 LDAP **Search** Request Protocol Op Base DN: cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com Scope: 0 (baseObject) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Done Message ID: 9 LDAP **Search** Result Done Protocol Op Result Code: 32 (No Such Object) Matched DN: ou=Finance,ou=Corporate,dc=ca,dc=com
10	Req	LDAP **Unbind** Request Message ID: 10 LDAP **Unbind** Request Protocol Op
	Resp	null[1]

2.2.2 Terminology in Stateful Service Virtualization

A service is consumed by the client (*i.e.*, SUT in service virtualization environment) through sending a request message to the service provider and receiving the response back from the provider. Thus, in service virtualization environment, the SUT sends request messages to the dependant services and gets the responses back from the

[1]Server does not send any response for the **unbind** request as the client wants to disconnect from the server.

dependant services. We refer the pair of a request and the corresponding response message as an *interaction* and a collection of consecutive interactions is referred to as an *interaction trace*.

A service usually defines the structure or format of the messages that the service receives and emits, and often involves different types of messages with different structures. The message structure is usually defined in terms of a sequence of *message fields*. These fields can be divided into those with values that are fixed across the same type of messages, and those with values that vary from message to message. The fixed fields are referred to as *message keywords*, have a special meaning as defined by the underlying message format, and carry structural information. On the other hand, the fields that differ from message to message are referred to as *value fields* and their content as *payload*.

As an example, consider the third request message (*i.e.*, **req#3**) in Table 2.1, an **add** request message of the Lightweight Directory Access Protocol (LDAP). It contains a number of keywords (*i.e.*, fixed fields across the **add** request messages) such as "**LDAP Add Request Message ID:**", "**cn=**", "**ou=**", "**mobile:**", "**sn:**", "**dc=**" etc. Among those keywords **ou=**, **sn:** and **dc=** represent an *organizational unit*, a *surname* and *domain component* respectively. Their corresponding value fields contain an organisation unit's name (*e.g.*, **Finance**), an individual's surname (*e.g.*, **MAJOR**) and a domain component's name (*e.g.*, **ca**) respectively.

In the context of this thesis, the concept of a keyword is similar to but broader than its usual meaning in machine languages. That is, *a keyword* is a maximum consecutive sequence of characters that is part of the format for a given message type. The keyword may be required (*i.e.*, occur in all messages of the given type) or optional (*i.e.*, occur in some of the messages of the given type). According to our definition, a consecutive sequence of keywords is considered as one keyword, and the delimiters used in message formats (if any) are regarded as part of the adjacent message keywords. For example, "**Request Message ID:**" in Table 2.1 is considered as one keyword in this paper's context. Note that this generalization of the keyword concept does not impact on the understanding or formulation of messages.

Ideally, every session starts with a **Bind** request and ends with an **Unbind** request in LDAP. Any service (*e.g.*, CA IM) usually refers to the data storage in formulating responses for the incoming requests. For example, the CA IM service creates a new entry or *record* in its data storage in responding to an **add** request and generates response based on the outcome of the operation. Similarly, the service removes a record specified in the request message from its data storage in responding to a **delete** request and the response is generated based on whether

the service is able to remove the specified record successfully or unable to find it in the data storage. Furthermore, a **search** request is used to retrieve the information matching the search query and the service finds the matched *records* upon receiving the **search** request and then puts a sequence of *message* fields in the generated response through breaking each of the matched records into a sequence of message fields. It indicates that the service (both stateful and stateless) breaks the records in its data storage into message fields while communicating with the client and the observation is valid for any record-based information service (*e.g.*, Banking service, Twitter, and GoogleBooks). Thus, a message (request or response) can be considered as having a sequence of *message fields*, one or more fields are used to identify a record in the service. We define such fields as *key fields* and values for those fields as *key payloads*. For example, each record in the identity manager (IM) service [1] is uniquely identifiable with the key field **dn** (distinguished name), *i.e.*, the second request message in Table 2.1 (**req#2**) is identified with "**cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com**".

2.3 Service Virtualization for Stateful Services

Service virtualization involves the creation and deployment of "virtual" services that emulate the specific behavior of the dependent services. As the responses of a stateful service depend on the service state as well as the incoming request messages, the service models should consider the service state in generating responses and consequently, providing a proper testing environment for the SUT.

In this section, we discuss three important issues of service virtualization concerning the emulation of stateful services. The main objective of this thesis is to create service models for stateful services through extracting relevant contextual and dependency relationships from the service interaction trace. To do so, we identify the different types of messages that are involved in the interaction, extract the formats from the different messages and infer the dependency relationship between messages and the correlation among message fields.

2.3.1 Format for Request Messages

As explained in Section 2.1.1, an appropriate response needs to be generated for each of the incoming request messages in the service virtualization environment to

facilitate the testing of SUT without accessing the actual services. Usually, a service supports different types of requests and may generates different types of responses for each type of request message. Therefore, the service models should able to identify different types of request messages that are involved in the communication, so the service models can generate appropriate responses for the incoming request messages. For example, Table 2.1 shows ten interactions (*i.e.*, ten requests and their corresponding response messages) between a client and the CA IM service [1]. As the interaction trace shows, the client sends a **`bind`** request for authentication and then sends **`add`**, **`search`**, and **`delete`** requests to add an entry, search for an entry and delete an existing entry respectively and finally, sends an **`unbind`** request to disconnect from the IM service. Thus, the interaction trace contains *five* types of request messages and the IM service responds with proper responses for each of the request types. As such, the service model needs to identify different types of request messages for generating the appropriate responses and consequently, to accurately simulate the behavior of the actual services.

Moreover, the response messages usually contain payloads and some of the payloads in the response messages may correlate with the payloads of the respective request messages. For example, the ***message ID*** field in LDAP messages is a unique identifier and is used to correlate the request and responses. All response messages generated for a request message must contain the same message ID as in the request message. The payload for the ***message ID*** field is "1" in both **`req#1`** and **`resp#1`** (Table 2.1). Therefore, a service model has to incorporate such correlations among message fields, which requires the isolation of the message keywords from the payloads *i.e.*, inferring the formats from the request messages to separate the payloads and analyzing the extracted payloads.

Furthermore, each request message has a number of message fields and has its own format. A number of message keywords can be identified for each type of request messages. For example, the third request message (**`req#3`** in Table 2.1) has thirteen message keywords including "**`LDAP Add Request Message ID:`**", "**`LDAP Add Request Protocol Op dn: cn=`**", "**`sn:`**", "**`title:`**", "**`,dc=`**", "**`mobile:`**" with "`3`" as the associated payload for the first keyword (*i.e.*, **`LDAP Add Request Message ID:`**) and "`Dominic MAJOR`" for the second keyword (*i.e.*, **`LDAP Add Request Protocol Op dn: cn=`**). On the other hand, the **`Unbind`** request message (**`req#10`** in Table 2.1) has only two keywords: "**`LDAP Unbind Request Message ID: `**" and "**`LDAP Unbind Request Protocol Op`**". The first keyword has an associated payload "`10`" whereas the second keyword does not have a payload.

From the above example, we note that not only do different types of request

messages have a different number of keywords, but also the keywords themselves differ from message type to message type. So, the extraction of formats from the different types of request messages requires the identification of request type from the request messages and then extraction of the keywords for each different type of request messages respectively.

2.3.2 Format for Response Messages

As described in the previous sub-section, a service usually support different types of request messages. But, a service not only supports different types of requests, it generates different types of responses for different types of requests. Moreover, a service may generate different types of responses for the same request at different times based on the service state. So, a model service has to identify the different types of responses that are generated for each type of the request messages. Different from the request messages, response messages do not always contain a message type field to indicate the type of responses, rather, each type of response message has its own set of keywords and has its own format. For example, the response of a **`Search`** request (**`resp#2`** in Table 2.1) contains six message keywords, whereas another **`Search`** response (**`resp#7`**) contains twenty seven message keywords. This suggests that the keywords of the responses for the same type of request may differ from response message to response message. Thus, the formats for the response messages are required to isolate the payloads from the messages and consequently, identify the correlations among message fields.

Unlike request messages, a response message may contain a *repetitive* sequence of message fields and hence, payloads of those message fields appear more than once in the response messages. For example, the name of an organizational unit (*i.e.*, "**`Finance`**") appears twice in the response message of a **`Search`** request (**`resp#7`** in Table 2.1) and it may appear multiple times depending on the number of entries returned in the search response. Thus, the payloads of such message fields can easily be mistaken as the message keywords (based on the occurrences) and hence, requires an explicit technique to identify the message keywords from the response messages. Moreover, a response message may contain *repetition* sequences of keyword-payload, *i.e.*, the whole message structure can be repeated multiple times based on the number of entries returned. For example, the LDAP search response, **`resp#4`** in Table 2.1 contains one search result entry followed by the *search result done* whilst **`resp#7`** contains two search result entries followed by the *search result done*. It indicates, different types of search responses may contain different numbers of entries depending

on the search query and the scope in the search request. Therefore, such *repetitions* of message structure (*i.e.*, keyword-payload sequence) need to be considered in format inference to accurately identify the message format.

2.3.3 Service Behavior

As described in Section 1.2, the responses of a stateful service depend on the service state as well as the incoming request, and the service state can be changed due to the previous sequence of interactions (*i.e.*, message dependency). Moreover, some of the message fields of the responses may correlate with the fields of the incoming request messages (*i.e.*, dependency among message fields). Thus, the service model need to identify and utilize such dependency relationships in formulating responses more accurately for the incoming request messages. We discuss below the key issues of identifying the dependency among messages and the generation of payloads for the message fields.

All requests and responses in Table 2.1 (except the **bind** request) are executed on two key payloads, *i.e.*, `cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com` and `cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com`. As the example interactions in Table 2.1 shows, the same type of **search** requests generates two different types of responses (*i.e.*, " the **No Such Object**" response for **req#2** and the "**Success**" response for **req#4**) at different times. This suggests that the service may generate different types of responses at different times based on the service state and the state itself can be modified through executing a request or a sequence of requests. For example, the second and fourth **search** requests (*i.e.*, **req#2** and **req#4**) with the same key payloads (*i.e.*, same **dn**) in the message fields generate different responses and an **add** request appears in between those **search** requests. It reveals that the service state has been changed due to the **add** request. Thus, an interaction or a sequence of interactions may affect or alter the responses for the following requests (*i.e.*, dependency among messages).

Again, based on the interactions in Table 2.1, the same request executed on different *key payloads* generates different responses. For example, the **Add** request with the key payload `cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com` (*i.e.*, **req#5**) generates "**Entry Already Exists**", while the same **Add** request with the key payload `cn=Susana LOW,ou=Finance,ou=Corporate,dc=ca,dc=com` (*i.e.*, **req#6**) generates "**Success**" response. It indicates that the state of the service for the key payload "**cn=Dominic MAJOR,ou=Finance, ou=Corporate,dc=ca,dc=com**" is different than the state for the key payload "**cn=Susana LOW, ou=Finance,ou=Corporate,dc=ca,dc=com**".

In addition to the dependency among messages, some of the message fields of the generated responses depend on the key payload (*i.e.*, record in the service) or the message fields of the corresponding request message (*i.e.*, data dependency). The generated response may contain two different types of payloads: i) message-describing payloads and ii) record-specific payloads.

Message-describing payloads are used to describe the message itself and may have correlation with the request messages. For example, the payload of the `message ID` field in the response message is the same as that in the corresponding request messages (Table 2.1).

The generated response may contain record-specific payloads in some of the fields. For example, the fourth response in Table 2.1 (*i.e.*, `resp#4`) contains a `mobile number`, `email`, `title`, `description surname`, etc., which are information related to the record (*i.e.*, key payload) "`cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca, dc=com`". Thus, the synthesized response should contain valid payloads in such fields to generate responses accurately.

2.4 General Requirements for Stateful Service Virtualization

In the previous section, we have discussed the issues of stateful service virtualization. In summary, a service supports different types of requests and each of them may generate different types of responses. Each type of message (request and response) has its own set of keywords and has its own format. Moreover, the responses of a stateful service have a dependency on the preceding sequence of interactions and some of the message fields of the response messages may correlate with the corresponding request or a sequence of the preceding requests. In this section, we present three general key requirements of creating "virtual" services for emulating stateful services based on the analysis in the above section.

REQ1: It is required to identify the *different types* of request messages as the different types of requests generate different types of responses. Moreover, the message fields in the request messages may correlate with the fields in the response messages. The payloads need to be isolated from the message keywords (*i.e.*, structure) to extract such correlations among message fields. Therefore, the formats of the request messages need to be extracted to synthesize responses accurately with appropriate payloads in the message fields.

REQ2: The service model has to identify different types of responses for the

different types of requests and even the same request, and infer the formats from the responses to further analyze the payloads. The service not only supports different types of requests, but may also generate different responses at different times for the same type of request, especially in stateful services due to the service state. So, the service model needs to identify the different types of responses for the same request to analyze and determine the reasons for generating different types of responses and consequently, to formulate the responses more accurately for the incoming request messages. Similar to the request messages, the payloads and the message keywords need to be isolated from each other (even for the messages with *repetitive* structures) for identifying data dependency. Therefore, the service model needs to extract the format for each type of response message.

REQ3: The service keeps changing its state while executing the requests from the client and the responses for the subsequent requests depend on the changed or updated service state. As the sequence of interactions is liable in determining the responses for the forthcoming requests, the service model has to identify such dependency among messages (*i.e.*, control dependency) for synthesizing responses accurately. Moreover, the payloads of the response messages depend on the payloads of the corresponding request or a sequence of request messages (*i.e.*, data dependency). Thus, the service model needs to identify such data dependency between message fields to determine the appropriate payloads in the response messages.

2.5 Summary

In this chapter, we have presented an overview of software testing in both traditional and service virtualization environments. We also have analyzed the issues of stateful service virtualization and identified key requirements based on these analyses. In summary, stateful service virtualization requires the extraction of the formats for both the request and response messages, and the identification of dependency relationships among messages and message fields. The identification of different types of request messages and extraction of formats for each request type are required to detect the incoming request messages and hence, to synthesize responses accurately. Again, it is required to identify the different types of responses that are generated for different request and the same type of request at different times. The formats of the response messages for each type of response messages need to be extracted to further analyze the payloads. The inference of the service behavior model is required to capture the observed individual message dependency relationship and synthesize responses based on the behavior of the service. We will analyze the existing literature

in the next chapter from the viewpoint of these key requirements.

Chapter 3

Related Work

In the previous chapter, we have presented the motivation of this thesis using the interaction trace collected from real-world services and identified the key requirements for creating stateful service models. In summary, the "virtual" replication of stateful services needs to i) extract formats for the request and response messages, and ii) identify dependencies between messages (*i.e.*, message dependency) and between message fields (*i.e.*, data dependency). In this chapter, we review and critically analyze the existing research works from the aspect of these key requirements.

This chapter is structured as follows. In Section 3.1, we present the traditional techniques in providing testing environments and their limitations in providing proper testing environments for the system under test (SUT). We discuss the existing service virtualization techniques and their limitations in virtualizing stateful services in Section 3.2. In Section 3.3, we present a review and discussion of existing research on extracting message formats and identify the research gaps. In Section 3.3, we review the existing research works on inferring behavior models that characterize the dependency relationships among messages and message fields. In Section 3.5, we review the existing research works on generating responses from the service virtualization viewpoint, while the actual purposes of these works were different, such as detecting malware in network security applications. The chapter is summarized in Section 3.6.

3.1 Background: Traditional Testing Environments

A suitable test-bed environment is the primary requirement for testing an enterprise system and hence, ensuring its quality. In this section, we present the traditional techniques of providing such environments through mimicking the specific behavior of the dependent components and replicating the physical devices.

HORUS [21] is a technique to generate stubs for marshaling and Unmarshalling the data values in multi-language Remote Procedure Calls (RPC). A different programming language supports different types of data and different machines (environments) use different conventions for storing data in memory (*e.g.*, Big-endian, and Little-endian). The proposed technique generates a language and machine-independent common specifications through deriving knowledge from a given set of language and machine specifications. Later, mock objects [4] and fake [5] simulate the behavior of the complex, real objects and provide support for running unit tests, especially when it is impractical or impossible to access the real objects. A number of mocking techniques is available including jMock [4], EasyMock [22] and DynaMock [23] to mimic certain behavior during the test. However, mock objects require an intimate knowledge of internal components as it demands the re-implementation of some of its components for each testing scenario.

On the other hand, another group of techniques has been proposed to create a proper testing environment by replicating physical devices and running multiple virtual machines simultaneously. Hardware virtualization can be divided into two categories: i) *full system virtualization*, where the architecture of hardware is replicated virtually entirely and ii) *paravirtualization*, where the modified version of operating system runs concurrently with other operating systems. The operating system "VM/370" developed by IBM provides an isolated and independent computing system to its multiple users. Lately, *Paravirtualization* systems such as VMWare [6], VirtualBox [7], Plex86 [24] etc. handle the virtual memory more efficiently compared to the full system virtualization. Such paravirtualization systems provide an isolated environment for its users. Such hardware virtualization techniques facilitate the testing of the interconnected systems by deploying the dependent services on virtual machines. However, such virtualization techniques (full or paravirtualization) through replicating the physical devices suffer scalability problems [8].

Instead of virtualizing hardware level, operating system-level virtualization, *i.e.*, containerization is another technique of providing isolated runtime environments for

applications. A number of containerization techniques have been proposed including Docker [25], Openvz [26], Google lmctfy [27] etc. Containerization involves bundling an application/system together with its dependent services and running on the host operating system. But a container is a light version of the operating system, which runs inside the host system. Thus, containerization suffers a similar scalability problem as in hardware virtualization techniques [8].

Summary: The existing efforts on mimicking the interaction behavior using stubs or mock objects when creating testing environments are tightly coupled with the programming language of the system under test. Moreover, it is time-consuming as some of its components need to be implemented repeatedly for each new testing scenario. On the other hand, the existing hardware virtualization techniques (full or paravirtualization) and containerization techniques require significant resources (*i.e.*, physical devices) to scale the testing environment and hence, unable to provide scalable testing environment for the SUT, *e.g.*, supporting tens of thousands of dependent services in a testing environment [28]. Therefore, the existing techniques, either by replicating physical devices or by replicating certain features of the deployment environment, are unable to provide a scalable and realistic testing environment for an enterprise system. Service virtualization provides a viable alternative for creating a proper testing environment by emulating the behaviors of the dependent components or services.

3.2 Service Virtualization

The concept of service virtualization [19] is to accelerate the development and improve the accuracy of the enterprise system, and reduce the overall cost. Service virtualization involves the creation and deployment of "virtual services" that emulate the specific behavior of the dependent components or services and facilitate the testing of the SUT without requiring access to the actual services. The formulation of such service models in the service virtualization environment is relatively more manageable than in mocking objects where its internal components require re-implementation for every testing scenario.

There is a number of commercial service virtualization tools that are available including CA Service Virtualization [29], ServiceV Pro [30], and IBM Service Virtualization [31]. Several open-source service virtualization tools are also available such as WireMock [32], hoverfly [33], Citrus [34], SoapUI [35], and Wilma [36]. Some research works have also been done for providing such a testing environment through creating the "virtual" replica of the actual services. The creation of virtual services

can be done in two ways: (i) manually defining service models by an expert with the required knowledge of underlying services [28], and (ii) automatically infer service models through extracting the relevant knowledge from the service interaction traces and utilizing them in generating responses [2, 11, 10, 9, 37]. In this section, we first discuss the limitations of the available commercial and open-source tools. Then, we present the existing research efforts on virtualizing services and discuss their limitations in virtualizing stateful services.

CA Service Virtualization [29] models the behavior of the actual service and the service model facilitates the testing and development of an enterprise system. It supports the creation of virtual services from i) the recorded service interactions when the actual service is available and accessible, and ii) Web Services Description Language (WSDL) when the actual service is unavailable. It provides support for a wide range of protocols across the front-end, middleware, and back-end technologies including SOAP, REST, HTTP, HTTPS, JDBC, etc. ServiceV Pro [30] provides support to virtualize REST and SOAP APIs, TCP, JDBC etc. It also allows the creation of virtual services either from the recorded service interactions or WSDL or API definitions. IBM service virtualization [31] comprises two products: *Rational Integration Tester*, a tool for setting up interfaces and configuring their behavior and *Rational Test Virtualization Server* to run the defined interfaces. It also provides support for web services (*e.g.*, REST, SOAP), middleware (*e.g.*, JMS) and back-end (*e.g.*, JDBC) protocols. On the other hand, open-source tools support web service mocking. For example, SoapUI [35] offers mocking SOAP and REST services, while WireMock [32] simulates HTTP-based APIs. However, none of the existing tools, *i.e.*, either commercial or open-source tools, consider the service state in creating virtual services. As such, these tools are unable to model the behavior of stateful services accurately.

The use of a scalable emulation environment has been presented in [28] to enable the realistic and comprehensive testing of an enterprise system. It uses a deterministic finite state machine (DFSM) to model the protocols of an enterprise system, where the input and output alphabets of DFSM are composed of a set of input and output commands that are available in the underlying services. The deduced DFSM can approximate the behavior of the actual services and can be used for emulation with the additional required information about the message encoding strategy of the modeled services. But, an expert with sufficient knowledge about the modeled service is required to define the service models manually, which is not always possible due to the lack of the information especially for legacy systems [38]. Moreover, it is time-consuming and error-prone as it requires to define the service models for

each testing scenario and for each service that is involved with the SUT [11].

To resolve this issue, several techniques [2, 9, 10, 11] utilize algorithms from bioinformatics to infer the service models automatically from the service interaction traces and do not require the explicit knowledge about the service to be modeled. *Whole cluster* [2] models the behavior of the services automatically without requiring explicit knowledge of the target services. It computes the dissimilarity scores for the incoming request by comparing it with the recorded requests using a variant of the Needleman-Wunsch [15] algorithm. The closest matched request/interaction is selected based on the dissimilarity score. Then, the symmetric fields are identified, *i.e.*, the fields contain the same data value in both the request and corresponding response messages. Finally, the responses are generated for the incoming requests through substituting data values (*i.e.*, payloads) for the symmetric fields in the synthesized responses. But, the proposed whole cluster technique generates many invalid (different than expected and malformed) responses due to picking the wrong interaction as the closest interaction. If the request messages of different types contain similar payloads, the proposed technique selects an incorrect interaction as the closest matched interaction and hence, generates the wrong response for the incoming request. Moreover, it is very inefficient in generating responses as it takes a long time to find the closest matched request, especially when the size of the recorded interaction trace is large. The entropy weighted approach [10] finds the closest matched interaction more accurately by putting more weights on the operation/request type field relative to other message fields. It performs an entropy analysis on the recorded interaction trace and then uses the entropy weighted Needleman-Wunsch algorithm to find the closest matched interaction. As the entropy weighted approach finds the closest interaction more accurately compared to the *whole cluster* approach, it synthesizes responses more accurately. However, inefficiency in the response generation is still not resolved as it also compares the incoming request against all the messages in the trace.

Cluster centroid [9] is another technique to generate service responses from the service interactions, especially to improve the runtime efficiency in response generation. It clusters the interactions based on the dissimilarity ratio matrix and selects an interaction as the centroid interaction for each cluster. The incoming request messages are now compared with the centroid interaction instead of comparing them with every interaction in the recorded trace, which significantly improve the runtime performance in generating responses. But, the incoming requests are often matched with the wrong interaction as it compares the request with the centroid interaction only and discards all other interactions, which leads to generating more

incorrect responses compared to the whole cluster approach. Opaque Service virtualization (OSV) [11] is the most effective and efficient technique so far in supporting the emulation of dependent services. It formulates a consensus prototype for each cluster, *i.e.*, for each type of request messages instead of selecting a centroid interaction in the *Cluster centroid* approach. It compares the incoming request with the generated prototypes, which is fast and more accurate compared to the previous approaches. Thus, it synthesizes responses more efficiently for the incoming request messages. However, all these approaches generate responses by considering the incoming request messages only and do not consider the service state in generating the responses. But the responses of a stateful service depend on the service state as well as the incoming request messages. Therefore, all these approaches are unable to synthesize responses accurately for stateful services.

To the best of our knowledge, the only prior technique to synthesize responses for stateful services is the one proposed in [37]. It employs classification based virtualization (CBV) and sequence-to-sequence based virtualization (SSBV) techniques to virtualize stateful services. In CBV, the responses of different types are labeled with a sequence of the request messages of length k. Then, a decision tree classifier is trained to learn the mapping between the requests and the corresponding class labels (response types). At runtime, for each incoming request message, it encodes the request type with k preceding interactions (request and response types) and then, compare with the training dataset to predict the response to send for the incoming requests. But, the proposed CBV technique requires the user-defined parser to identify request types, response types, and contents from the interactions. On the other hand, the SSBV technique uses Long Short Term Memory (LSTM), a special neural network to model the services and uses Tensorflow [39] in learning the network. At first, a vocabulary is created with the letters and characters of the interactions. The input trace is transformed into a list of enumeration IDs, which is encoded into a vector. Finally, a decoder is used to decode the vector into the output sequence. But, the proposed technique has three major limitations. First, it requires the user-defined parser to identify the type of requests and the type of responses from the traces. So, an expert with sufficient knowledge about the encoding of the messages is required for applying the technique. Second, it does not consider the generation of payloads or data values for responses. Usually, the response for any request either in stateful or stateless services contains a considerable amount of data values especially the responses for *Read* requests in *CRUD* services. Third, it does not consider the data or record (entity) specific service state in formulating responses. Usually, the state of the service can be different for each record or entity

that the service maintains. For example, in a banking service, an account can be active, frozen or closed, and the outcome of any operation (*e.g.*, withdraw, deposit, etc.) on that account depends on the current state for that account. Similarly, the response of any operation for another account depends on the state of that account and the service state is unique for each account. As such, the applicability of this approach is limited in virtualizing stateful services.

Summary: The existing service virtualization techniques including the commercial software, open-source tools and research prototypes do not consider the service state in creating "virtual" services. As such, the generated responses by the service models only consider the incoming request messages. But, the responses of a stateful service depend on the service state in addition to the incoming request messages. Thus, none of the existing techniques can generate accurate responses for stateful services and hence, these techniques are limited in simulating the behavior of stateful services accurately. However, the only technique considering the service state in generating responses [37] has major limitations as described above, and can not be adopted in practice for virtualizing stateful services.

Moreover, none of the existing techniques consider the structure the messages in synthesizing responses, especially in substituting the payloads from the incoming requests to the synthesized responses. It leads to breaking the message structure and resulting in malformed responses, *i.e.*, non-conformant to the actual service protocol. Again, some of the message fields of the generated response may correlate with the corresponding request message or a sequence of the preceding requests. But, none of the existing techniques consider such correlations in synthesizing responses. Only a few techniques described in [2, 9, 10, 11] consider *symmetric* correlations, *i.e.*, correlation between the request and corresponding response messages. But, such symmetric correlations are extracted only from the matched interaction without considering the structure of the messages, *i.e.*, by comparing string/characters between the request and corresponding response messages. As such, the extracted symmetric correlations are not generic, *i.e.*, not applicable to all interactions, especially when the length of payloads in the matched interaction is different than the incoming request. Therefore, the correlations among message fields without considering the message structure leads to synthesizing invalid responses.

To generate more accurate responses for the stateful services by considering the service state, the formats of the messages and the relationships between messages and between message fields (*i.e.*, the behavioral model) need to be extracted from the service interactions. The service behavior model and the formats of the messages can be utilized in formulating responses more accurately. We discuss the existing

research works on extracting message formats, inferring the service behavior model from message traces, and generating responses for incoming requests, in the following sections.

3.3 Message Format Extraction

One of the key requirements of our thesis is to extract the formats of the messages in order to derive the service models more accurately and consequently, to generate more accurate responses for incoming request messages. The problem of extracting message formats has been investigated previously with various approaches having been proposed. In general, the existing message format extraction approaches can be divided into two broad categories: (i) program code based approaches and (ii) message trace based approaches.

3.3.1 Program Code Based Approaches

In this section, we present the existing research works on inferring message formats of the messages by instrumenting the program code of the underlying protocols or services. The manual effort of reverse engineering is time-consuming and error-prone. A number of such reverse-engineering projects targeting different protocols such as Samba protocol [40], MSN Messenger protocol [41], Yahoo! Messenger protocol [42], have been lasted for a long period in extracting the specification of the protocols. Moreover, the extracted specifications in such projects are protocol-specific. Automatic extraction of message formats is required as the protocols frequently update its details to provide new functionality.

There are several approaches that infer formats of the messages automatically from the executable code. Polyglot [43] uses dynamic analysis of program binaries to infer message format by analyzing how the program processes and operates on the protocol data. Prospex [44] infers message format based on the message trace and the behavior of the server for each message in the trace. It runs the application in a controlled environment that supports dynamic data tainting [45, 46, 47] for recording all operations to the messages and generates an execution trace for further analysis. Once the execution trace is generated, it uses the technique proposed by Wondracek *et al.* [48] to infer the message format. The proposed technique in [48] splits the messages in the execution trace into fields by identifying the length and delimiter fields. Once the messages are decomposed into fields, it derives additional four types of information from the messages, *e.g.*, fields that have a special meaning in

the protocol and fields that are used as file names. Finally, the sequence alignment algorithm is applied on the extracted fields, *i.e.*, the structural information about individual messages, to infer the message format. The technique proposed in [49] extracts output data formats such as file formats and network packet formats by analyzing the executable binary. AutoFormat [50] is a tool to infer message format by analyzing the processing of different fields of messages by the binary code. The intuition of the approach is that different fields of the same messages are managed in different execution context, for example, different run time, different call stack etc. AutoFormat considers two types of execution context: the run-time call stack and the location of the instruction being executed. Then, it builds a protocol field tree after identifying fields from the execution context. Finally, the Backus-Naur Form (BNF) specification is derived as the format from the protocol field tree. Unlike the existing message format inference techniques that consider only deciphered messages, ReFormat [51] focuses on inferring formats from the encrypted messages. Similar to the AutoFormat, it also collects the execution trace that contains the run-time call stack. Then, it divides the execution trace into different execution phases, *i.e.*, into an encryption phase and a decryption phase based on the cumulative percentage of arithmetic and bit-wise instructions executed between transition points. After that it considers the *lifetime* [52] of the data to identify the result of the functions for encryption. Finally, it identifies the buffer containing the decrypted messages by analyzing the lifetime of the data, and AutoFormat [50] is adopted to infer the format of the decrypted messages.

Summary: These research efforts infer formats of the messages by analyzing the program code that implements the underlying protocols or services. These approaches generate an execution trace to capture the behavior of the program on messages. Then, formats of the messages are inferred based on the execution contextual information. They require access to the source code or binary code. But, such access to the source code or binaries is not always possible and consequently, these approaches are not applicable when the program code is unavailable, which is not uncommon when proprietary or legacy systems are involved.

3.3.2 Message Trace Based Approaches

In this section, we present related work about message format extraction from message traces. This category of methods infer message formats from the message traces only and do not require any access to the source or binary code. Existing research on extracting message format from message traces can be grouped into two categories

according to the method used in extracting keywords:

- Tokenization Based Methods
- n-gram Based Methods

We introduce each of these categories in turn.

3.3.2.1 Tokenization Based Methods

Tokenization based methods split the messages into tokens using the separators and/or delimiters in the messages. The natural delimiters and separators include tabs, space(s), special characters, etc.

Protocol Informatics (PI) [53] is a tool for inferring message format from the message trace. It identifies the common parts and variable parts through optimal alignment of messages. It applies the Needleman-Wunsch [15] sequence alignment algorithm to align the byte sequence of messages and the distance matrix is created based on the local sequence. The alignment result is then used to build a phylogenetic tree [1] using Unweighted Pairwise Mean by Arithmetic Averages (UPGMA) [54]. Finally, progressive alignment (*i.e.*, tree traversing rules) is used to perform multiple sequence alignment and identify the common/fixed portions across messages.

Discoverer [55] reassembles IP packets into textual messages and then, uses a predefined set of delimiters to divide the messages into tokens. It adopts a recursive clustering approach on tokenized messages to group the messages into clusters of messages with each cluster having the same format. First, tokens are clustered based on the classes (binary or text) and the direction of the messages. Second, a recursive clustering is applied to identify the Format Distinguisher (FD) field by inferring a format for each cluster and then, the inferred formats are examined token by token from left to right. It selects candidate FD fields based on the threshold (*i.e.*, maximum distinct values for FD tokens), which leads to over-classification. Thus, the formats across the sub-clusters are compared and similar formats are merged using type-based alignment. The structure (*i.e.*, token type, token sequence) of two message formats are compared in type-based alignment. It breaks the messages with a set of delimiters based on the assumption that some delimiters are used to differentiate the fields in messages. But, not all protocols use delimiters to separate fields and such delimiters may not be known even if they are used in messages of unknown or legacy protocols.

[1] Phylogenetic tree is an evolutionary tree, which shows evolutionary relationships among organisms.

ASAP [56] is a framework to analyze the messages and classify them based on the similarity of the tokens. It breaks the messages into a set of tokens using the predefined delimiters. It extracts the alphabet of strings (*i.e.*, tokens[2] for textual protocols and n-grams[3] for binary protocols) to characterize the traffic. The network payloads are then mapped to a vector space that reflects the characteristics of the extracted alphabets. Then, the communication template or message format is identified as the sequence of tokens corresponding to the base directions of the alphabets in the vector space. ASAP analyzes the payloads to discover the patterns in honeypot data and helps in designing an intrusion detection system. However, the inferred message format is imprecise [57].

ReverX [58] is a tool to derive the message format as automata. Similar to Discoverer, it uses the predefined delimiters to break the messages into tokens. First, it filters out the messages to make sure the trace contains messages of interested protocol (*i.e.*, the trace may contain messages of multiple protocols and trace is filtered so it contains messages of one protocol). Then, messages in the message trace are grouped into application sessions, where sessions are identified based on the same source and destination IP address and port number, TCP sequence number and temporal gaps between messages. After that, the messages of each session are split into a set of tokens based on the delimiters. Then, it generates a partial language automaton (PTA), which accepts the observed sequence of tokens in the message trace. Equivalent states are merged based on the appearances of the data values associated with fields and the inferred automata models the formats of the messages.

Wang *et al.* [59] proposed a technique to extract message formats based on keyword extraction. First, it breaks the messages into tokens using the predefined delimiters. Low frequent tokens are then filter out using Jaccard index [60] and an initial FSM is built by scanning each message using the extracted tokens, which accept all messages in the trace. Finally, the Moore reduction procedure [61] is applied to generalize the initial FSM with a minimum number of states and transitions as the format of the messages.

Maatta and Raty [62] adopted the Message Sequence Chart (MSC) to model the sequential activities of legitimate network traffic to detect intrusions in traffic. It uses predefined protocol-specific features to create a feature element for converting a single message to the modeled XML. The XML prototype of legitimate network traffic is generated through searching some predefined fields (*i.e.*, IP address, source

[2] a set of all strings separated by delimiters
[3] a set of all strings with fixed length n

port, destination port, etc) in the messages. Then, the MSC notation is used to model the communication scenario. This method is not applicable when such prior knowledge about the protocol is not available or only certain message traces of the protocol are accessible.

SANTaClass [63, 64] is a technique to automatically group the network traffic into clusters based on packet payload content (PPC). It generates a signature from the traffic in combination with a real-time traffic classifier. It identifies the common terms that are present in multiple messages as signatures instead of identifying all keywords from messages. SANTaClass extracts statistical signatures from the packet payloads of message trace. Finally, it generates Prefix Tree Acceptor (PTA) with transition probability as the inferred signatures for the messages.

The approach presented in [65] focused on mining protocol keywords from the WebSocket [66] protocol. First, it extracts candidate keywords considering a keyword usually appears more frequently across messages and does not appear many times in one message. Messages are then quantized as a sequence of alternate payload and keyword tokens. A Hidden semi-Markov Model (HsMM) is built to capture the relationships between the length of a data field and the preceding keyword. It captures both the temporal and spatial position relations,*i.e.*, how keywords appear in messages. Finally, the true keywords are singled out by maximum likelihood estimation, *i.e.*, a keyword by its positions in the optimum state sequence.

Summary, these techniques infer the format of the messages after identifying tokens (*i.e.*, keywords) from the messages. These techniques break the messages into tokens using a set of predefined separators and/or delimiters. All of the above techniques except Discoverer infer a single format from the messages and do not consider different types of messages that have different formats. On the other hand, although Discoverer considers clustering the messages based on the message type, it does not consider the keyword/token inaccuracy issue. Particularly, payloads appear more frequently in messages with *repetitive* structure, which is very common in the service responses (*i.e.*, false keywords) and keywords in shorter lengths can be covered with longer length keywords. Thus, these techniques are not effective in extracting formats from the messages of a service.

3.3.2.2 *n*-gram Based Methods

Another group of existing techniques finds string patterns of arbitrary length in messages, called n-grams, where n denotes the number of characters or bytes in the pattern. The techniques of inferring message formats using n-grams can be further divided into two categories: i) Inferring generic format, and ii) Inferring type-specific

format.

Inferring Generic Format

A group of existing approaches focuses on extracting message fields that are common to *all* messages. Veritas [67] considers the most frequently occurring strings as message structure information (message units). In order to obtain the high-frequency units, Veritas introduces the two-sample Kolmogorov-Smirnov statistical testing method (K-S test) [68] to tackle the resulting message units set. After the test, it extracts the message format by combining the message units from the ordered sequence. Biprominer [69] extracts binary protocol message formats from a given message trace by inferring a transition probability model as the message format. It first recursively identifies frequent binary patterns of arbitrary length, called *n*-grams (where n denotes the number of bytes in the pattern), in messages. Then, the probability of a keyword following another keyword is measured. Each keyword has a transition probability associated with other keywords. Each message is labeled with the identified keyword/cell. Finally, the messages with labeled patterns are converted into a transition probability model. The technique proposed in [70] focuses on the variable portions of the messages and identifies them by using a sequence alignment algorithm to align all messages in the message trace. It can find the offsets and lengths of the payload information only when all the messages share the same format. NEMESYS (Network Message Syntax Analysis) [71] infers the structure from network messages of binary protocols. It segments messages without comparing multiple messages to find similarities in byte values, rather it analyzes a single message at a time to discover its intrinsic structure. It uses the delta of the congruence in bit values of consecutive bytes, *i.e.*, Bit Congruence Delta (BCD) as the similarity measure to identify the field boundaries, where Bit Congruence (BC) is the bit-wise similarity of bytes [72]. Then, the standard Gaussian filter is used to find the overall tendency across multiple bytes of a message. Finally, messages are segmented with the inferred field boundaries by approximating the inflection points of Gaussian-smoothed deltas of Bit Congruences for the message field.

Summary, these techniques infer the format of the messages by identifying the fixed and variable portions of the messages. But, this group of techniques assumes that all messages in a message trace contain the same number of message fields and share the same format. As such, these techniques can only be applicable for the messages having a similar format. However, a service API usually has different types of messages with different formats. As such, their applicability in extracting message formats for service APIs is limited.

Inferring type-specific Format

AutoReEngine [73] and ProDecoder [74] aim to extract type-specific message formats form message traces. AutoReEngine adopts the Apriori algorithm [75] to identify keywords by breaking the messages into n-grams. Then, noisy keywords are removed by analyzing the variations of keywords' positions. Each message in the trace is scanned for identifying the frequent keyword series and messages are classified according to the similarity among keyword series. The message clustering is based on the intuition that different types of messages contain different sequences of message keywords. The keyword series in each cluster is regarded as the corresponding message format. However, as keywords are extracted *prior* to clustering, false keywords (*e.g.*, payload information that appears frequently) can be extracted as keywords, leading to the creation of more clusters than actual message types and inference of more formats than the actual formats.

ProDecoder [74] decomposes the input messages into *words* by using n-grams and then applies a Latent Dirichlet Allocation (LDA) based approach (inspired by natural language processing) to identify keywords. It computes the probability distribution of a keyword over n-grams instead of extracting actual keywords. Given the identified keywords and their corresponding probabilities, the Information Bottleneck (IB) algorithm is then used to cluster the messages. Finally, it infers the message format for each cluster using sequence alignment. Due to the difference between natural languages and machine-generated languages (*i.e.*, messages that are transferred between programs), directly applying LDA causes the inaccuracy of ProDecoder in message clustering. Often, different types of messages are put into a single cluster, consequently leading to format over-generalization.

Summary: The above approaches cluster messages based on the similarity among the keyword sequences and then infer the format for each cluster. But, the inaccuracy of keyword extraction leads to incorrect classification of messages. These approaches extract a large number of false keywords (payloads) especially from the messages containing *repetitive* patterns and missed actual keywords that are short in length and covered by longer keywords. Due to those inaccurate keywords, these techniques either create more clusters than the actual number of clusters or mixes multiple types of the message into a single cluster, which leads to improper classification of messages. Therefore, they infer imprecise formats of the messages and the approaches are not effective in separating payloads from the message keywords. Moreover, the response messages in services usually contain *repetitive* sequence of message keywords and payloads. Therefore, payloads appear more frequently compared to the message keywords in the response messages. As such, more inaccurate

keywords can be extracted by the above techniques, which leads to imprecise message clustering and format inference.

3.4 Service Behavior Inference

Another key requirement of our thesis is to discover the behavior of the service from the message traces. The service behavior consists of the dependency relationships between messages and between message fields. So that our approach can generate more accurate responses by utilizing the inferred dependency relationships (between messages and between message fields) in formulating responses. We consider two types of dependency relationships in the service behavior model: i) message dependency to express the dependency relationships among messages, and ii) data dependency to characterize the relationships between message fields. To the best of our knowledge, there have been no direct research works on inferring such dependencies among request and response messages. As such, we discuss the existing research works on inferring the control dependency from the system logs[4] to express a valid sequence of method calls and the data dependency in the following sub-sections.

3.4.1 Control Dependency Model

This section discusses the related works about inferring the control dependency model from the system logs that describes the valid sequence of inter-component method calls. Existing research on inferring such model can be classified into two categories according to the types of model inferred: i) automaton based [76, 77, 78, 79, 80, 81, 82, 83], and ii) non-automaton based, *i.e.*, temporal rule mining approaches [84, 85, 86, 87].

3.4.1.1 Automaton Based Techniques

A number of techniques have been proposed to derive the control dependency model from the system logs, which is used to validate the software system and monitor network applications including intrusion detection and prevention, traffic normalization etc. The model is inferred either in the form of Finite State Automata (FSA) [77, 79, 88, 83, 89] or in the form of Graph [80, 76, 82, 81], where states/nodes represent the method invocation and edges/transitions represent valid sequencing of

[4]System logs contain information about events that are happened during the execution of the system, whereas traces may contain input data to the system or the captured interactions between the system and a client.

method invocations. Most of the model inference works consider the method invocation sequence only, while a few of them incorporate data-flow information along with the method invocation sequences with aim to represent the behavior of the system more accurately. We present the existing research works on inferring the control dependency model of each of these categories in turn.

Method sequence only: A number of techniques infer the control dependency model from the observed method invocation sequences in the system logs, which is used to validate the system. kTail [77] is the most basic algorithm for inferring such model from the logs. It builds a prefix tree acceptor (PTA) from the logs by creating an individual state for each event (*i.e.*, method) in the logs. Then, it merges the equivalent states based on their *future states*. The states are considered as equivalent based on their k-tails states. Two states with the same k-future are considered as k-equivalent states. But, such state merging leads to an imprecise model as the states can be different in their distant future even if they are similar in the next future [90, 83]. kSteering [79] refines the state merging with the inferred temporal rules to improve the precision of the model. It mines the future-time [91, 92] and past-time [93] temporal rules from the traces and utilizes the inferred rules to "steer" the inference of the model and consequently, increase the precision of the inferred model.

Synoptic [80, 76], Perfume [81], and CSight [82] infer control dependency models from the system logs to capture the behavior of a system in the form of Graph. All these approaches create an initial coarse-grained model by creating only one node for each event (*i.e.*, method) instead of creating multiple nodes in a different path for each trace. These techniques use the CEGAR [94] approach to refine the model by utilizing the mined temporal invariants from the input traces. Synoptic mines three types of temporal invariants: i) "always followed by", "never followed by" and "always precedes by" compared to the past-time and future-time temporal rules in [79] to increase the precision of the inferred model. Synoptic uses the BisimH algorithm for refinement which starts with the initial model (compact) until it satisfies all the invariants mined from the trace graph (union of traces). The initial model is built by simply merging the same states of the trace graph, *i.e.*, every state will appear once in the initial model. The resulted model accepts all sequences of method calls that are present in the logs and may accept method invocation sequence that was not present in the logs. Each of the inferred invariants is converted into a small FSM using the BisimH algorithm that accepts the method invocation sequence satisfying the invariant and then the FSM is traversed to update the graph. The model checker returns a counterexample path if an invariant is not satisfied in the model graph.

After refinement, multiple partitions of the graph can be combined by preserving the invariants. Finally, the model graph is coarsened using the kTail algorithm with k=0. The model inferred by Synoptic satisfies the temporal properties that improved developer confidence in the correctness of their systems and were useful for finding bugs [95]. Perfume [81] extends Synoptic [80, 76] to include the resource utilization information in the system model to understand the system behavior and resource use explicitly. System logs may contain how much memory each method consumes, how much time taken to execute each method, etc. The inferred model using Perfume encodes those resource usage information to improve program analysis and software processes. Different from the above approaches, CSight [82] mines the logs of a concurrent system and infer the behavior of the system in the form of communicating finite state machine (CFSM). It infers the model for a concurrent system from the logs that satisfy the *happens-before* relation [96] and the inferred model can be used in debugging and verifying the concurrent systems.

Summary: These approaches are useful to infer the system's control dependency model as a sequence of method calls from the system logs. The inferred models summarize and generalize the inter-component method calls to support debugging and verifying the system [79]. The system logs contain the execution sequence of methods of each component of a system. As such, the inferred models from the logs are used to identify errors or incorrect order of method invocations in the system. Therefore, the newer techniques use more stricter generalization rules, *i.e.*, preventing imprecise state merging, using the mined temporal invariants to preserve the observed method invocation sequence in the logs.

These techniques are useful for determining the dependencies between messages. But, none of the above techniques is dedicated to predicting the response for the incoming request. Moreover, the proposed approaches only consider the method invocation sequence to infer the system model and ignore the complex interplay between the data values and the methods/requests. In the service context, the data values play an important role in determining the type of responses to send for the incoming requests.

Data-flow information with method sequence: Several techniques consider the data value with the method invocation sequence in model inference to represent the behavior of the system more precisely.

The gkTail algorithm [78] infers the model as Extended Finite State Automata (EFSA) from the system logs and the source code. The inferred model incorporates both the event/method sequences and the data values. The transition in EFSA is annotated by algebraic constraints. At first, it merges similar traces which share the

same event sequence but differ on the value assigned to parameters. Then, it mines the algebraic constraints from the value assigned in transitions and builds a PTA where edges/transitions are annotated with constraints. Finally, states with the same future of k-length are merged. CONTRACTOR++ [83] infers FSA from the execution traces [5] using the inferred state invariants (value-based invariants) [6]. It runs Daikon [97] to infer the program state invariants from the execution traces and then runs CONTRACTOR [89] to infer the FSA from the inferred invariants. State-enhanced k-tails (SEKT) [83] is another algorithm to infer FSA from the value-based invariants. Similar to CONTRACTOR++, it infers value-based invariants from the execution traces by running Daikon and then it applies the modified kTail [77] to infer the FSA by utilizing the inferred invariants. Different from the traditional kTail, SEKT allows merging the states only if states share the same value-based invariants in addition to the same *k*-future. Trace-enhanced MTS (Modal Transition System) Inference (TEMI) [83] is another algorithm to infer the FSA from the value-based invariants that represent the method invocation sequence of a system. Unlike CONTRACTOR++, TEMI prevents imprecise state merging based on the invariants. It runs CONTRACTOR++ in the first phase to infer the FSA with *maybe* transitions. Then, TEMI converts the *maybe* transitions to *required* transitions based on the observed transitions in the execution traces.

Specforge [88] is an approach to synergize the models from the different miners with the aim to increase the precision of the inferred model as the existing FSA inference techniques require improvement to make them practicable. It constructs N models from N different FSA inference techniques and then unifies these FSAs into a single model through model fission and model fusion. The FSAs are deconstructed as a list of constraints (temporal logic) between events/methods, *i.e.*, a set of Linear Temporal Logic (LTL) constraints [98]. Then, the SPIN model checker [99] is used to remove invalid LTLs, *i.e.*, LTLs that are rejected by the FSA. A subset of LTL is selected from the list of LTLs that are extracted from all FSAs using four heuristics (*i.e.*, union, majority, satisfied by, intersection). Each of the selected LTL is converted into a simple FSA and finally, Specforge combines the FSAs into a unified FSA in the model fusion stage.

An empirical study between techniques that infer a simple model and techniques that infer an extended model with information about data-values shows that adding data values with operations does not compromise the quality of the inferred model

[5] The execution trace includes detailed information about the actual parameter and the return value for each method.

[6] State invariants consist of method pre- and post-conditions and object invariants. Such invariants indicate which method can execute in a particular state

and the inferred model can represent the behavior of the actual system more accurately [100].

Summary: These approaches capture the relationship between data values and the method sequence. Therefore, the inferred model using these techniques can represent the behavior of the system more accurately compared to the model considering method invocation sequence only. These techniques instrument the source code to include the effect of data values in the method invocation sequence in the FSA. But, such access to the application program limits their applicability to infer such kind of dependency model where access to the application program is not possible. As such, these techniques are not directly applicable in service virtualization as only interactions between a client and the actual service are accessible and the source code that implements the underlying services is unavailable.

3.4.1.2 Temporal Rule Mining Approaches

In this section, we discuss the existing research works about temporal rules mining which capture constraints that a system under analysis must follow. Yang et al. proposed a technique for inferring temporal rules automatically by analyzing the execution traces [85]. To infer temporal properties, it instruments the target program by inspecting entry and exit points of each method. The execution traces are collected through running the instrumented program with a test dataset. Then, the candidate temporal rules are instantiated and the satisfaction ratio for each candidate pattern is computed from the execution traces. The candidate temporal rules with the strongest satisfaction ratio are selected as temporal invariants. Finally, the inferred temporal invariants are validated through a model checker or manual inspection. Ernst et al. proposed an approach to discover invariants, along with instrumentation named Daikon [84]. It instruments the source code and writes the variable values to the file for inferring the invariants. The file contains a set of values for every instrumented place (*i.e.*, at the beginning, loop head and end, and method start). A set of candidate invariants are detected through a range of testing, *e.g.*, a single variable is tested against its values and multiple variables are tested for identifying relationships among them. Finally, the invariants having higher confidence than specified by the user are selected as the temporal invariants.

The technique proposed by Henkel et al. discovers algebraic specifications [86] from the source code implemented in object-oriented languages such as Java [87]. It infers the method signature from the source code and infers algebraic specification in the form of axiom to characterize the behavior of the implementation.

Summary: These approaches infer invariants from the program code to verify

the correctness of the software or program. The objectives of these techniques are verifying, debugging the code, while the objective in this thesis is to emulate the services. However, the inferred temporal rules using these techniques are useful in refining the control dependency model.

3.4.2 Data Dependency Model

In a service environment, the generated responses for the incoming request messages usually contain payloads. The payloads for some of its fields may directly depend on the corresponding request message or the sequence of preceding messages. A data dependency model can be used to represent such kind of correlations among message fields. Therefore, the service models can generate responses with appropriate payloads for the incoming requests by utilizing such relationships. This section presents the related works about finding correlations among message fields.

The existing research works on virtualizing stateless services [2, 11, 10, 9] find symmetric fields, *i.e.*, message fields that share the same value in the request and corresponding response messages. With the identified symmetric fields, these techniques substitute the data values in the synthesized responses with the data values in the incoming request. This substitution ensures that the generated responses contain the same payloads as in the request message. The techniques proposed in [101, 102, 103] identify the *message id* field in the request messages through analyzing the data-values of the message fields across messages. The message fields are first checked for numeric test and if any such numeric field found, the data values of that field across the messages are checked whether the values are increasing or decreasing sequentially. PRISMA (Protocol Inspection and State Machine Analysis) [104] extracts a few correlations among message fields in the messages. It extracts the exact/matched fields, *i.e.*, fields that contain the same value in both the request and corresponding response messages, partially matched fields, *i.e.*, a part (front or back) of a field value from the request message is present in the response messages with or without separators.

Summary: The existing approaches of identifying the correlations among message fields are useful in generating responses with appropriate payloads in such correlated fields. In a service virtualization environment, the response message usually contains two types of payloads: i) message describing fields that describe the message itself and ii) record-specific fields that describe a record or an object managed by the service. Among message-describing fields, some of the fields may contain the same payloads in both the request and corresponding response messages. Ef-

forts in identifying symmetric field [2, 11, 10, 9] and PRISMA [104] are dedicated to identify such message-describing fields from the messages. Therefore, the symmetric fields are identified from the matched interaction (*i.e.*, one interaction only) by comparing the request and corresponding response messages. But, the extracted symmetric field correlations are not verified with the rest of the interactions and hence, an invalid correlation can be extracted that stands only for the matched interaction. Moreover, the message structure is not considered in extracting such correlations, which leads to synthesizing incorrect responses for the incoming request at times. On the other hand, PRISMA considers the message structure in finding such message-describing fields. But, it does not consider different types of messages having different formats. Moreover, the payloads for the record-specific fields may appear or change in preceding interactions.

3.5 Response Generation

The "virtual" services emulate the actual services through generating responses for the incoming request messages. We already have reviewed the existing service virtualization works, which generate responses for the incoming requests. In addition, there are some other research works that generate responses for different purposes, *e.g.*, detecting malware variant in network intrusion detection systems and preventing the spam email for spreading.

ScriptGen [102, 103] is a tool that generates "replay" for incoming requests through finding and updating certain fields of the recorded responses. It uses a byte-wise statistical analysis followed by the Needleman-Wunsch [15] algorithm to group similar messages into clusters. It then builds a state machine based on the sequence of messages observed in the trace. The edge represents incoming request messages and node represents response messages. The sequence alignment algorithm is applied to response messages for finding similar response messages and then the nodes having similar response messages are merged. Finally, a honeyd-compatible script is generated from the state machine to emulate the behavior of the protocol.

RolePlayer [101] is another tool to mimic the server behavior through generating "replay" for the incoming requests with the aim to detect malware variants. It parses the messages and finds the dynamic fields (*i.e.*, IP address, hostname) through aligning message pairs. It utilizes the knowledge of syntactic conventions of fields (*e.g.*, IP address contains "dotted quads") and contextual information (*e.g.*, domain names) in finding the dynamic fields. Once the dynamic fields have been identified, it uses a single message as a template and inserts the data values in the template

for the dynamic fields in generating a replay.

Summary: These approaches utilize sequence alignment to generate the replay for the incoming requests and able to emulate the server/protocol behavior. However, RolePlayer does not consider the dependency relationship between messages in generating replay. Hence, it is not particularly useful in emulating services in virtualization environments where responses depend on the previous sequence of messages. On the other hand, the state machine inferred by ScriptGen is too simple in terms of emulating services to provide a suitable testing environment for the SUT. For example, it does not consider that a request can generate different types of responses at different times. Moreover, the responses usually contain some payloads. None of these techniques consider the generation of payloads for the synthesized responses. As such, these approaches are limited for the purpose of generating responses in the context of service virtualization.

3.6 Summary

In this chapter, we have reviewed the existing research efforts that relate to service virtualization. We have investigated the traditional testing frameworks to support the testing of an enterprise system. Although the traditional techniques facilitate testing by replicating resources or mocking interaction behaviors, they are unable to provide a scalable and realistic testing environment for an enterprise system. We also have reviewed the existing service virtualization techniques and found their limitations in virtualizing stateful services as they did not consider the service state in formulating responses. We have also discussed the current state of the art in extracting formats of the messages. Although the existing techniques groups messages having the same type of messages in a single cluster and infer formats for each cluster, they have limitations. We have reviewed the existing control dependency model inference techniques and identified their limitations in inferring such model in emulating services. None of the control dependency model inference technique considers predicting responses for the incoming requests and does not consider the interplay between payloads and the messages. We also have analyzed the existing techniques for extracting data dependency model and none of them consider the message structure in extracting such dependency among message fields, which leads to identifying imprecise dependency relationships. Finally, we have reviewed the existing research works on generating responses for the incoming requests. It reveals the in-capabilities of the existing works in generating a response in a service virtualization environment as they did not consider different types of responses for the

different types of requests and even for the same request at different times. Moreover, the generation of payloads has not been considered in synthesizing responses. This review, did not find any research works that capably deal with stateful service virtualization, *i.e.*, considering the service state in generating responses for the incoming request messages.

Chapter 4

Format Extraction for Request Messages

In the previous chapter, we have identified the research gaps based on the key requirements of virtualizing services, especially emulating stateful services. In this chapter, we present our approach to fulfilling one of the key requirements, *i.e.*, infer the formats of the request messages to identify different types of requests and to separate the message keywords from the payloads. We identify the request type field of messages based on the observation that the request type field has lower variations compared to other fields and the messages of the same request type have the lowest dissimilarity between them. Then, the interactions are grouped into clusters based on the values of request type field and finally, the request message formats are inferred through extracting the type-specific keywords from the request messages and identifying the keyword pattern across the request messages in a cluster.

The rest of the chapter is organized as follows: In Section 4.1, we present the details of our approach to infer the request message formats from the request messages after giving an overview of the approach. Experimental results on several services and applications are reported in Section 4.2. Section 4.3 discusses the assumptions and limitations of our approach. Finally, we summarize this chapter in Section 4.4.

4.1 Approach

The target of our approach is to infer accurate formats for the request messages to separate the keywords from the payloads *i.e.*, to extract the message structure from the request messages. The key to achieving this goal is the ability to (i) accurately

Note that this chapter is the detailed version of the work presented in [105].

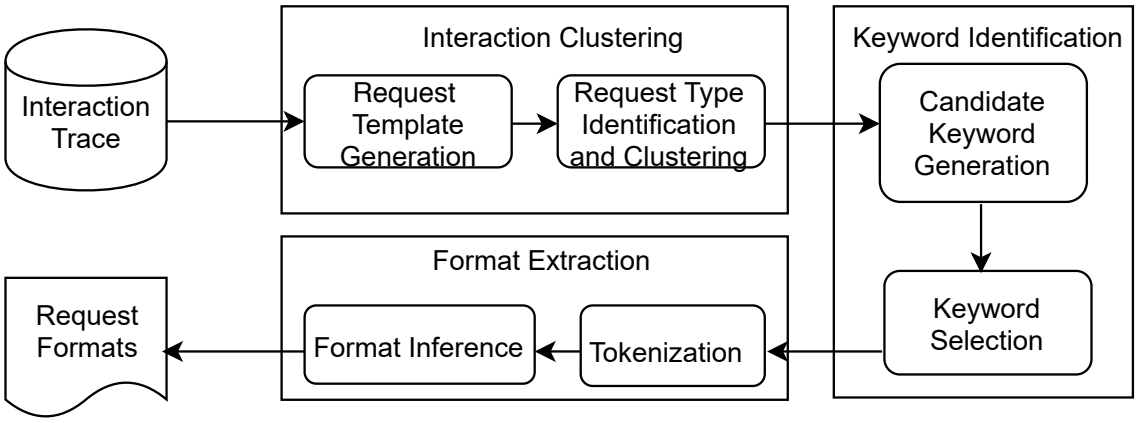


Figure 4.1: Overview of Request Format Extraction Approach

separate the interactions into request-type based clusters and (ii) accurately identify the keywords for each type of request message. The motivation behind our approach is that the request messages contain a field to indicate the request type, different types of request messages usually have a different number of keywords, and the keywords themselves differ from request message to request message. The request messages are clustered based on the observation that the request type field has the *least* variation across all request messages compared to the other fields and accurate clustering has the lowest dissimilarity among the request messages in a cluster. Then the request messages from the clustered interactions allow us to extract request type-specific keywords from the request messages and, hence, infer request message formats for each cluster.

Figure 4.1 shows the overview of our approach that includes three major steps: *interaction clustering*, *keyword identification*, and *format extraction*. In the first step (interaction ilustering), we group all interactions in an interaction trace into different request-type based clusters with the aim to have each cluster containing the interactions of one request type only. To do so, we first identify those parts of the request messages that are *common* to all request messages, *i.e.*, the *Common Request Message Template*. We then identify the request type field by stripping away the common parts and identifying the next level of least varying part across all request messages as the *request type* field. Finally, we divide the interactions into clusters according to the values of the request type field.

In the second step (keyword identification), we identify the message keywords from the request messages of each cluster, which are the constant parts across the request messages in that cluster (in contrast to their variable payload fields). We first break the request messages down into words of different lengths and select those words that have high frequency across the request messages in the cluster (according to a threshold) as *candidate keywords*. Then, we remove those candidate keywords

that are primarily parts of other candidate keywords and are not keywords on their own, resulting in the set of (true)*keywords* for the given cluster.

In the third step (format extraction), we convert each message in a cluster into a sequence of keywords with the aim to infer format through generalizing the keyword sequence across all messages in a cluster. Each request message in a cluster is first converted into a sequence of alternating keyword and payload fields (*i.e.* Tokenization), using the keywords identified in the previous step. Then, we apply the Synoptic tool [76] to infer the *request message format* for the cluster in the form of a finite state machine (FSM), and finally convert it into a regular expression.

Table 4.1 shows an example interaction trace collected by intercepting the communication between a client and the CA Identity Manager (IM) [1] service, which will be used to illustrate the detail of our approach to inferring the request message formats in the following sub-sections. The CA Identity Manager (IM) [1] is an enterprise-grade identity management service to manage the digital identity of the personnel of a large organization and to control the access of their resources. It uses the Light Weight Directory Access Protocol (LDAP) [20] to add and manage the user accounts. As Table 4.1 shows, the server does not send any response message for the **`Unbind`** request message as it is used to indicate that the client wants to disconnect from the server.

Table 4.1: Example Interaction Trace

No.		Interactions
1	Req	`LDAP` **`Bind`** `Request Message ID: 1 LDAP Bind Request Protocol Op LDAP Version: 3 Bind dn: cn=admin,dc=ca,dc=com Authentication Data: Authentication Type: Simple Bind Password: 1228013670`
	Resp	`LDAP` **`Bind`** `Response Message ID: 1 LDAP Bind Response Protocol Op Result Code: 0 (Success)`
2	Req	`LDAP` **`Add`** `Request Message ID: 2 LDAP Add Request Protocol Op dn: cn=Dominic MAJOR,ou=Finance,ou=Corporate,dc=ca,dc=com mail: Dominic.MAJOR@ca.com mobile: 6017515 description: Customer Service objectClass: inetOrgPerson title: Administrative Operator sn: MAJOR cn: Dominic MAJOR`
	Resp	`LDAP` **`Add`** `Response Message ID: 2 LDAP Add Response Protocol Op Result Code: 0 (Success)`

Table 4.1 Continued: Example Interaction Trace

No.		Interactions
3	Req	LDAP **Add** Request Message ID: 3 LDAP Add Request Protocol Op dn: cn=Susana LOW,ou=Management,ou=Corporate,dc=ca,dc=com mail: Susana.LOW@ca.com mobile: 6726920 description: Customer Service objectClass: inetOrgPerson title: Applications Administrator sn: LOW cn: Susana LOW
	Resp	LDAP **Add** Response Message ID: 3 LDAP Add Response Protocol Op Result Code: 68 (Entry Already Exists)
4	Req	LDAP **Add** Request Message ID: 4 LDAP Add Request Protocol Op dn: cn=Dominic RAYMOND,ou=Record,ou=Customer,dc=ca,dc=com mail: Dominic.RAYMOND@ca.com mobile: 6817147 description: Applications Administrator objectClass: inetOrgPerson title: Customer Service sn: RAYMOND cn: Dominic RAYMOND
	Resp	LDAP **Add** Response Message ID: 4 LDAP Add Response Protocol Op Result Code: 32 (No Such Object) Matched DN: ou=Customer,dc=ca,dc=com
5	Req	LDAP **Search** Request Message ID: 5 LDAP Search Request Protocol Op Base DN: cn=Dominic RAYMOND,ou=Record,ou=Customer,dc=ca,dc=com Scope: 0 (baseObject) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Done Message ID: 5 LDAP Search Result Done Protocol Op Result Code: 32 (No Such Object) Matched DN: ou=Customer,dc=ca,dc=com
6	Req	LDAP **Search** Request Message ID: 6 LDAP Search Request Protocol Op Base DN: ou=Finance,ou=Corporate,dc=ca,dc=com Scope: 1 (singleLevel) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Entry Message ID: 6 LDAP Search Result Entry Protocol Op dn: cn=Clive BRANCH,ou=Finance,ou=Corporate,dc=ca,dc=com cn: Clive BRANCH mail: Clive.BRANCH@ca.com mobile: 6312753 description: Design Administrator objectClass: inetOrgPerson title: Financial Economist sn: BRANCH LDAP Search Result Done Message ID: 6 LDAP Search Result Done Protocol Op Result Code: 0 (Success)

Table 4.1 Continued: Example Interaction Trace

No.		Interactions
7	Req	LDAP **Search** Request Message ID: 7 LDAP Search Request Protocol Op Base DN: ou=Construction,ou=Projects,dc=ca,dc=com Scope: 1 (singleLevel) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Entry Message ID: 7 LDAP Search Result Entry Protocol Op dn: cn=Eddy BRYCE,ou=Construction,ou=Projects,dc=ca,dc=com cn: Eddy BRYCE mail: Eddy.BRYCE@ca.com mobile: 5940538 description: Software Consultant objectClass: inetOrgPerson title: Communications Services Co-ordinator sn: BRYCE LDAP Search Result Entry Message ID: 7 LDAP Search Result Entry Protocol Op dn: cn=Gwen HUNTER,ou=Construction,ou=Projects,dc=ca,dc=com cn: Gwen HUNTER mail: Gwen.HUNTER@ca.com mobile: 6340642 description: Response Engineer objectClass: inetOrgPerson title: Purchasing Consultant sn: HUNTER LDAP Search Result Done Message ID: 7 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
8	Req	LDAP **Search** Request Message ID: 8 LDAP Search Request Protocol Op Base DN: ou=Training,ou=Human Resources,dc=ca,dc=com Scope: 1 (singleLevel) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:
	Resp	LDAP **Search** Result Entry Message ID: 8 LDAP Search Result Entry Protocol Op dn: cn=William SIMPER,ou=Training,ou=Human Resources,dc=ca,dc=com cn: William SIMPER mail: William.SIMPER@ca.com mobile: 6813842 description: Computing Officer objectClass: inetOrgPerson title: Consulting Technician sn: SIMPER LDAP Search Result Entry Message ID: 8 LDAP Search Result Entry Protocol Op dn: cn=Joseph GRIMES,ou=Training,ou=Human Resources,dc=ca,dc=com cn:Joseph GRIMES mail: Joseph.GRIMES@ca.com mobile: 6953740 description: Training Officer objectClass: inetOrgPerson title: Industrial Clerk sn: GRIMES LDAP Search Result Done Message ID: 8 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
9	Req	LDAP **Search** Request Message ID: 9 LDAP Search Request Protocol Op Base DN: ou=Industrial Relations,ou=Customer,dc=ca,dc=com Scope: 1 (singleLevel) Deref Aliases: 3 (derefAlways) Size Limit: 0 Time Limit: 0 Types Only: false Filter: (objectClass=*) Attributes:

Table 4.1 Continued: Example Interaction Trace

No.		Interactions
	Resp	LDAP **Search** Result Entry Message ID: 9 LDAP Search Result Entry Protocol Op dn: cn=Brad DUFFY,ou=Industrial Relations,ou=Customer,dc=ca,dc=com cn: Brad DUFFY mail: Brad.DUFFY@ca.com mobile: 8219206 description: Hardware Support objectClass: inetOrgPerson title: Acting Engineer sn: DUFFY LDAP Search Result Done Message ID: 9 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
10	Req	LDAP **Unbind** Request Message ID: 10 LDAP Unbind Request Protocol Op
	Resp	

4.1.1 Interaction Clustering

The purpose of this step is to identify the request type fields in the request messages of the interaction trace, and cluster the interactions according to the request type. For the example interaction trace in Table 4.1, the interactions should be clustered into four different groups according to their request types, that is, one group for each of the request types (highlighted in bold): **Bind**, **Add**, **Search**, and **Unbind**.

In most existing approaches, messages are clustered according to multiple message fields (keywords) and their co-occurrence relationships in a message [53, 55, 58, 73, 74, 106]. Based on this observation, we further utilize the fact that each request message has a field that defines the request type, *e.g.*, the message field containing **Bind**, **Add**, **Search**, or Unbind in Table 4.1. Therefore, in this step, we focus on identifying this request type field before using its values to cluster the interactions. It involves two major sub-steps. First, we identify the common message template for request messages that contain all message fields that are common to all request messages. These common message fields are to be excluded from consideration in the next sub-step, as they cannot be used to differentiate the different types of request messages. Second, we identify the specific message field that contains the request type and then use it to cluster the interactions. The overall clustering algorithm is shown in Algorithm 1 and is further discussed in detail below.

Algorithm 1: Interaction Clustering

Input: threshold T, message set M, wildcard character W
Initialize: field extractor set $regex = \emptyset$, entropy set $entropy = \emptyset$, dissimilarity set $d = \emptyset$, message clusters $clusters = \emptyset$
$P \leftarrow$ GenerateMessageTemplate(M) based on [11]
for $i \in \{1, \ldots, length(P)\}$ **do**
  if $(P(i) = W)$ **then**
    for $j \in \{i+1, \ldots, length(P)\}$ **do**
      if $P(j) \neq W$ **then**
        $P(i..j-1) \leftarrow (.*?)$
        break
      end if
    end for
  end if
end for
$temp \leftarrow \text{Split}(P, (.*?))$
for $i \in \{1, \ldots, |temp| - 1\}$ **do**
  $regex \leftarrow regex \cup \text{Concat}\ (temp(i), (.*), temp(i+1))$
end for
for $i \in \{1, \ldots, |regex|\}$ **do**
  $values(i) \leftarrow \emptyset$
  for $m \in M$ **do**
    $val \leftarrow$Extract$(m, regex(i))$
    if IsEmpty(val) **then**
      $values(i) \leftarrow \emptyset$
      break
    end if
    $values(i) \leftarrow values(i) \cup \{val\}$
  end for
  $entropy(i) \leftarrow$ CalcEntropy$(values(i))$ from Eq. (4.2)
end for
for $i \in \{1, \ldots, |entropy|\}$ **do**
  if $entropy(i) \leq T$ **then**
    $clusters(i) \leftarrow$GetClusters$(values(i), M, regex, i)$ from Eq. (4.3)
    $d(i) \leftarrow DissimIndex(clusters(i))$ from Eq. (4.4)
  end if
end for
$typeIndex \leftarrow \arg\min_i\{d\}$
Return $clusters(typeIndex)$

Request Template Generation

The purpose of this step is to generate a common template from the request messages with the aim to separate the variable and fixed message fields. In this sub-step, at line *3* of Algorithm 1, we identify the common request template that captures all those message fields that appear in every request message. These common message fields cannot be used to differentiate different types of request messages and are, therefore, excluded from further consideration. We adopt the message template generation technique from [11] to identify the common fields. It uses the multiple sequence alignment (MSA) technique [107] and inserts gaps (denoted by the '◊' symbol) into the messages during the alignment process, to obtain the MSA profile. For example, Figure 4.2 shows the character frequencies in the alignment result for the example interaction trace in Table 4.1

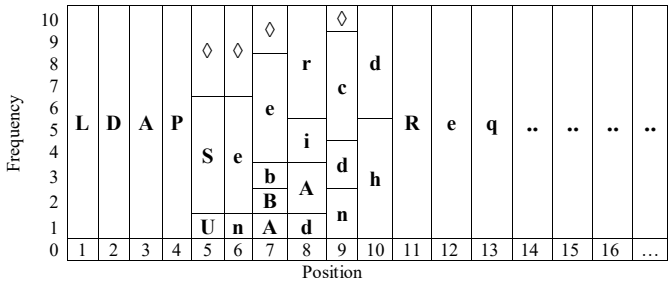


Figure 4.2: Character Frequencies in the Alignment Result

Then, the common request message template, P, which contains all the common message fields across all request messages, is calculated by iterating through each byte position of the MSA profile, and calculating a template symbol, p_i, at that position using the following Equation [11]:

$$p_i = \begin{cases} c_i, & \text{if } q(c_i) \geq f \wedge c_i \neq \Diamond \\ \perp, & \text{if } q(c_i) \geq \frac{1}{2} \wedge c_i = \Diamond \\ \#, & \text{otherwise} \end{cases} \tag{4.1}$$

where c_i is the consensus symbol at position i, $q(c_i)$ denotes the relative frequency (over the number of messages) of the consensus symbol c_i (*i.e.* the symbol with the highest frequency count at that position across all request messages), f the relative frequency threshold, '◊' the gap symbol, '#' the wildcard symbol, and '⊥' the truncation symbol. After a template symbol is calculated for each position, all truncation symbols are deleted from the common message template, as each of them represents a position filled with mostly gaps. The common request template then contains two categories of symbols: the consensus symbol with frequency above the

threshold at the given position, and the wildcard symbol at other positions that do not have a consensus symbol above the threshold. With a threshold of or close to 1.0, the message template contains the message fields common to all request messages, separated by the wildcard symbol.

By applying this method with a frequency threshold $f = 0.80$ (as described in [11]), for example, the following common request template is generated for the request messages given in the Table 4.1 (please note that the space character is shown as '␣'):

```
LDAP␣######␣Request␣Message␣ID:␣#␣LDAP␣######␣Request␣Protocol␣Op␣##:␣
##=####################,dc=ca,dc=com␣####:␣#################
```

Request Type Identification and Clustering

The purpose of this step is to identify the request type by analyzing the variable portions of the common request template. In this sub-step, we consider only those parts of the request messages that are not included in the common template to identify the request type field. In doing so, we utilise the observation that a service supports a limited number of request types, and thus the request type field has lower variation than other fields across the request messages. To identify the request type field, we need to extract the actual values of the non-common message fields for all request messages. From line *4* to *13* in Algorithm 1, we first replace each variable field in the common message template (*i.e.*, those fields containing consecutive wildcard symbols) with the pattern `(.*?)`. Then, in line *14*, we split the modified common message template into a sequence of the fixed (or common) and variable fields. From line *15* to *17*, we generate *field extractors* in the form of regular expressions by merging every variable field with its two surrounding fixed fields. With the above example common template, for example, Table 4.2 shows the generated *field extractors* using Algorithm 1 (lines *4* to *17*).

Now, we extract the actual data values for each variable field using each of the field extractors in Table 4.2. With lines *18* to *27* in Algorithm 1, for each of the field extractors, we extract all the values of its corresponding variable field across all request messages. For example, Table 4.3 shows the extracted values (separated by ',') from the example interaction trace (Table 4.1) using the field extractors in Table 4.2.

After extracting all the variable field values, we exclude some of the fields from further consideration based on the values for the fields. As Table 4.3 shows, some request messages contain *null* or *empty* value for certain message fields in Table 4.2 and therefore, those fields are excluded as the value for the request-type field

Table 4.2: Inferred Field Extractors

Field No.	Field Extractors
1	`LDAP␣(.*?)␣Request␣Message␣ID:␣`
2	`␣Request␣Message␣ID:␣(.*?)␣LDAP␣`
3	`␣LDAP␣(.*?)␣Request␣Protocol␣Op␣`
4	`␣Request␣Protocol␣Op␣(.*?):␣`
5	`:␣(.*?)=`
6	`=(.*?),dc=ca,dc=com␣`
7	`,dc=ca,dc=com␣(.*?):␣`
8	`␣:␣(.*?)$`

Table 4.3: Extracted Values for Message Fields

Field No.	Values
1	Bind, Add, Add, Add, Search, Search, Search, Search, Search, Unbind
2	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3	Bind, Add, Add, Add, Search, Search, Search, Search, Search, Unbind
4	LDAP Version, dn, dn, dn, Base DN, Base DN, Base DN, Base DN, Base DN,
5	cn, cn, cn, cn, cn, ou, ou, ou, ou,
6	admin, Corporate, Corporate, Customer, Customer, Corporate, Projects, Human Resources, Customer,
7	Authentication Data, mail, mail, mail, Scope, Scope, Scope, Scope, Scope,
8	1228013670, Dominic MAJOR, Susana LOW, Dominic RAYMOND, (objectClass=*) Attributes:, (objectClass=*) Attributes:, (objectClass=*) Attributes:, (objectClass=*) Attributes:, (objectClass=*) Attributes:, 10 LDAP Unbind Request Protocol Op

can not be *null* or *empty*. For example, **`req#10`** does not contain any value for the fourth message field (Table 4.2) and hence, the fourth field is excluded to be considered as request type field. Similarly, we exclude the fields *5, 6* and *7* from further consideration. After that, we calculate the randomness or entropy of the values for the remaining fields using the Shannon Index entropy $H(i)$ [108] in line *28* of Algorithm 1.

$$H(x) = -\sum_{i=1}^{R} Pr(i) \times ln(Pr(i)) \tag{4.2}$$

where $H(i)$ is the Shannon entropy for the values of the i-th variable field and is between 0 (constant) and $ln(n)$ (perfectly random), where n is the number of

distinct values of the variable field, $Pr(j)$ is the probability or relative frequency of j-th distinct field value in the value set. For example, Table 4.4 shows the entropy and the number of distinct values of the remaining message fields (*i.e.*, four fields), where a lower entropy value indicates a more stable variable message field.

Table 4.4: Entropy of the Example Dataset in Table 4.1

Field No.	Entropy	No. of distinct values
1	1.17	4
2	2.30	10
3	1.17	4
8	1.50	6

Table 4.5 reports the entropy for one of the evaluation datasets, a more comprehensive dataset (LDAP) as the example interaction trace (Table 4.1) contains only 10 interactions and does not show reasonable variations among message fields.

Table 4.5: Entropy of LDAP Dataset in Section 4.2.1

Field No.	Entropy	No. of distinct values
1	1.80	8
2	4.84	231
3	1.80	8
4	4.98	311
5	5.44	2066

With the observation that the request type field has the *least* data variation among all the variable fields, we select the variable field that has the least entropy (variation) as the request type field to be used for interaction clustering. However, there may be service protocols where certain message fields may have even less variability than the request type field. For example, some messages in a particular protocol may have a field that can take either "`SUCCESS`" or "`FAIL`" as possible values. To include the actual request type field for consideration, we select a number of candidate (request type) fields that have the relatively lowest variations, by choosing an entropy threshold (line *31* of Algorithm 1). Based on an examination of common service protocols, we set the entropy threshold to 3.40, which allows maximum 30 variations in request type fields.

We further stipulate that the most accurate clustering has the lowest dissimilarity among the request messages in a cluster. Hence, we cluster the interactions according to the distinct data values of each of the candidate fields (line *32* of Algorithm 1).

Equation 4.3 defines the method for separating the interaction set M into clusters.

$$\begin{aligned} GetClusters(V, M, regex, i) =& \{c_j | j = 1..|V|\} \\ \text{where } c_j =& \{m | \forall m \in M \wedge \\ & Extract(regex, m)_i = V_j\} \end{aligned} \tag{4.3}$$

where V is the vector of possible values for the ith field extracted using the regular expression *regex*.

We then calculate the dissimilarity among request messages for each clustering outcome (line *33*), and choose the clustering outcome with the lowest dissimilarity (lines *36* and *37*). The dissimilarity index for the clusters of each candidate field is calculated according to mean cluster dissimilarity given in Equation 4.4.

$$DissimIndex(C) = \frac{1}{|C|} \sum_{c_i \in C} \frac{\sum_{m_1 \in c_i} \sum_{m_2 \in c_i} \Delta(m_1, m_2, w_M)}{|c_i|^2} \tag{4.4}$$

where C is a set of interaction clusters, Δ is the entropy weighted Needleman-Wunsch edit distance [15], and w_M is the vector entropy weights calculated for request message set $M = \bigcup_{c_i \in C} c_i$ using the method of [10].

For the LDAP example trace given in Table 4.1, the chosen request type field is the first variable field[1], which contains the field values of `Bind`, `Add`, `Search`, and `Unbind`, and has the lowest entropy for field values and the lowest dissimilarity for its clusters, respectively. Table 4.6 shows the clustering result of interactions in Table 4.1

Table 4.6: Interaction Clustering Result

Cluster	Interaction No.
Cluster 1 (`Bind` cluster)	1
Cluster 2 (`Add` cluster)	2, 3, 4
Cluster 3 (`Search` cluster)	5, 6, 7, 8, 9
Cluster 4 (`Unbind` cluster)	10

4.1.2 Keyword Identification

The purpose of this step is to identify the keywords that appear in the request messages of a given cluster. The input to this step is the set of the request messages

[1]Field 3 has exactly the same value and is also a valid request type field for the example interaction trace. We choose the first variable field as the request type field.

Algorithm 2: Candidate Keyword Generation

1: *Input:* minimum length N, threshold T, message set M
2: *Initialize:* keyword tuples $K(keyword,\ frequency) = L(keyword,\ frequency) = \emptyset$, keyword multiset $W = \emptyset$
3: $n \leftarrow N$
4: Boolean $found \leftarrow true$
5: **while** $found = true$ **do**
6: **for** $m \in M$ **do**
7: **for** $y_i \in \{x_i..x_{i+n-1} \in m | i = 0..length(m)\text{-}n\}$ **do**
8: $W \leftarrow W \cup \{y_i\}$
9: **end for**
10: **end for**
11: $L(keyword, frequency) \leftarrow$ CountFrequency(W)
12: $found \leftarrow false$
13: **for** $k(keyword, frequency) \in L$ **do**
14: **if** $frequency \geq T \times |M|$ **then**
15: $K \leftarrow K \cup \{k\}$
16: $found \leftarrow true$
17: **end if**
18: **end for**
19: $n \leftarrow n + 1$
20: **end while**
21: *Return* K

as identified by the clustering described in the previous subsection, and the output is a set of keywords for the corresponding cluster of request messages. This step has two major sub-steps. First, a set of candidate keywords are generated from the request messages of a cluster. Second, remove the candidate keywords that are substrings of other keywords and, therefore, are not independent keywords themselves.

Candidate Keyword Generation

In the first sub-step (see Algorithm 2), each request message in the cluster is broken down into words (or grams) of length n using the n-grams technique (lines *5* to *10*), and then the frequencies of the generated distinctive words or n-grams are calculated (line *11*). We set the initial value of n to be 2, so that we can identify all keywords with a minimum of 2 characters. We then select only those candidate keywords that have a relative frequency (over the message set size) greater than a threshold value T (lines *13* to *18*). At line *19*, the length of the grams n is incremented by *1*, and the process continues to find grams of length $n+1$, until no new candidate grams are found for a given length. As we apply this candidate keyword identification step for each request-type based message cluster and the

request messages in a single cluster are supposed to share the same format, we set the threshold T to 1.0, to select as candidate keywords only those words that appear on average at least once in each request message.

Algorithm 2 returns a set of candidate keywords and their frequency of occurrence in the message trace. For example, Table 4.7 shows some candidate keywords and their frequencies, which are generated by Algorithm 2 for the request messages of the request type `Add` cluster from Table 4.1.

Table 4.7: Candidate Keywords

No.	Candidate Keyword	Frequency
1	`LDAP Add Request Protocol Op dn: cn=`	3
2	`LDAP Add Request Protocol Op dn: cn`	3
3	`Request Protocol Op dn: c`	3
4	`LDAP Add Request Message ID:`	3
5	`LDAP Add`	6
6	`LDAP`	6

Keyword Identification

The candidate keyword list contains actual keywords and some of their substrings; a substring of a frequent string is also frequent. Thus, in the second sub-step, we need to remove those candidate keywords that are actually substrings of other candidate keywords. For example, in Table 4.7, candidate keywords number 1 and 4 are actual keywords whereas the remaining candidate keywords are substrings of these keywords and hence, can potentially be removed from the candidate keywords set. However, if a substring k' of a candidate keyword k occurs "in isolation", that is, at a different location/position in a request message from k and not enclosed by another candidate keyword, then k' cannot be removed and needs to be retained.

Algorithm 3 describes the process that is used to remove candidate keywords when they occur as a substring of a candidate keyword (longer), but retain them when they occur "in isolation" (as described above). At line *2*, we sort the candidate keywords in descending order according to their length. For a particular candidate keyword $k1$ (line *3*), we identify other candidate keywords $k2$ that are a substring of $k1$ (lines *4* and *5*) and reduce $k2$'s frequency by the frequency of $k1$ (line *6*). After this process, a candidate keyword's frequency reflects the number of its distinctive occurrences, *i.e.*, excluding its occurrences as a substring of other candidate keywords. Table 4.8 shows the candidate keywords after frequency subtraction of the candidate keywords in Table 4.7.

Algorithm 3: Keyword Identification

1: *Input:* threshold T, keyword list K
2: Sort K according to keyword length in descending order
3: **for** $k1(keyword1, frequency1) \in K$ **do**
4: **for** $k2(keyword2, frequency2) \in K$ **do**
5: **if** $k1 \neq k2$ AND $keyword1$ contains $keyword2$ **then**
6: $frequency2 \leftarrow frequency2 - frequency1$
7: **end if**
8: **end for**
9: **end for**
10: **for** $k(keyword, frequency) \in K$ **do**
11: **if** $frequency < T \times |M|$ **then**
12: $K \leftarrow K \backslash \{k(keyword, frequency)\}$
13: **end if**
14: **end for**
15: *Return* K

Table 4.8: Candidate Keywords After Frequency Subtraction

No.	Candidate Keyword	Frequency
1	`LDAP Add Request Protocol Op dn: cn=`	3
2	`LDAP Add Request Protocol Op dn: cn`	0
3	`Request Protocol Op dn: c`	0
4	`LDAP Add Request Message ID:`	3
5	`LDAP Add`	0
6	`LDAP`	0

At lines *10* to *14* in Algorithm 3, we select only those remaining candidate keywords that have a relative frequency greater than the threshold as actual keywords. For example, Table 4.9 shows the actual keywords extracted from the candidate keywords in Table 4.8.

Table 4.9: Extracted Keywords from the Candidate Keywords in Table 4.8

No.	Keywords
1	`LDAP Add Request Protocol Op dn: cn=`
4	`LDAP Add Request Message ID:`

4.1.3 Format Extraction

The purpose of this step is to use the set of keywords identified in the previous step and the original request messages for each request-type based cluster, to extract the

request format for that cluster. The output message format for each request-type based cluster is presented in the form of a *regular expression*, *i.e.*, a sequence of alternate keywords and payload information segments. This step has two sub-steps: (1) Tokenization, and (2) Format Inference.

Tokenization

In *Tokenization*, the original request messages are split into a sequence of tokens, where each token either corresponds to a keyword that was identified in the previous step, or to payload information. This is done by selecting a keyword from the identified keyword lists, finding it in the request messages and storing its position(s) in the messages. This process continues until all keywords are considered. All the payload portions of the request messages are then labelled as "***VARIABLE***". For example, we obtain the following sequence for a LDAP **Add** request message (second interaction in Table 4.1) after the tokenization step:

LDAP␣Add␣Request␣Message␣ID:␣***VARIABLE***␣LDAP␣Add␣ Request␣Protocol␣Op␣dn:␣cn=***VARIABLE***,ou=***VARIABLE***,ou=***VARIABLE***,dc=ca,dc=com␣mail:␣***VARIABLE***@ca.com␣ mobile:␣***VARIABLE***␣userPassword:␣***VARIABLE***␣description:␣***VARIABLE***␣objectClass:␣inetOrg-Person␣title:␣***VARIABLE***␣sn:␣***VARIABLE***␣cn:␣***VARIABLE***

Format Inference

In the second sub-step, we use the Synoptic tool [76] to infer the message format from the above tokenized messages. In general, Synoptic infers a finite state machine (FSM)-based system model that expresses the correct sequences of events from an event log. For our purpose, we treat the individual tokens (keywords and "***VARIABLE***") in a message as events. Given the tokenized messages of each cluster as input, Synoptic produces a finite state machine in terms of the tokens of these messages. For example, Figure 4.3 shows the inferred format of the **Add** messages from the CA Identity Manager (IM) service. It's important to note that Synoptic *generalizes* the individual message instances into a pattern including unlimited repetitions, *e.g.*, the iterative occurrences of "***VARIABLE***" and '**,ou=**' (highlighted in red in Figure 4.3) commonly found in API and programming language definitions.

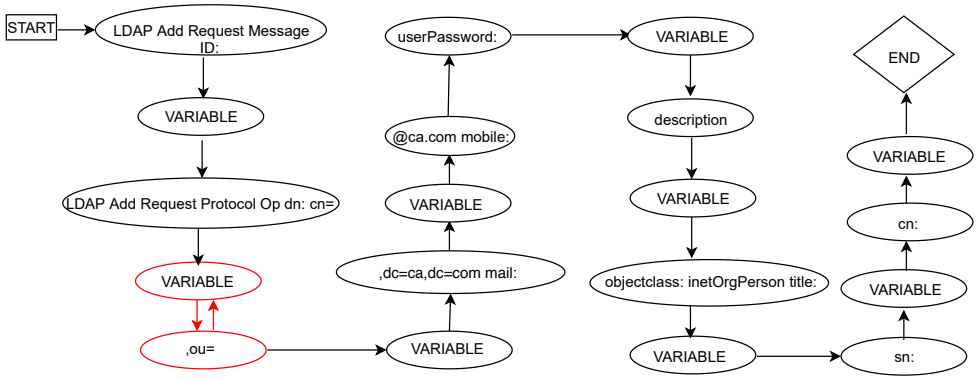


Figure 4.3: Format of the ADD Message of CA Identity Manager Service [1]

By replacing "***VARIABLE***" with '`.*`', we can convert a FSM into a regular expression. For example, the FSM in Figure 4.3 is converted into the following regular expression as the format of the **`Add`** messages in Table 4.1 (space character is shown as '␣'):

```
LDAP␣Add␣Request␣Message␣ID:␣.*␣LDAP␣Add␣ Request␣Protocol␣Op␣dn:␣cn=
(.*,ou=)+.*,dc=ca,dc=com␣mail:␣.*@ca.com␣mobile:␣.*␣userPassword:␣.*
␣description:␣.*␣objectClass: ␣inetOrgPerson␣title:␣.*␣sn:␣.*␣cn:␣.*
```

4.2 Evaluation

We have evaluated our approach to inferring message formats and compared it to two state of art approaches (*i.e.*, ProDecoder [74] and AutoReEngine [73]), by applying them to the interaction traces of real-world services. We use 10-fold cross-validation approach [109] to the experimental datasets *i.e.*, a given dataset is split into 10 folds where each fold is used as a testing dataset and the remaining folds are used as a training dataset. In the following subsections, we describe the datasets used, introduce the evaluation metrics, and present the results of our experiments.

4.2.1 Datasets

We have applied all the three approaches to the following four interaction traces collected from real services that used the following APIs or protocols:

- **CA IM (LDAP):** Lightweight Directory Access Protocol (LDAP) is commonly used for accessing and maintaining distributed directory information services over the Internet Protocol [20]. It is a binary protocol that uses the ASN.1 [110]

encoding to encode and decode text-based message information to/from its binary representation. An enterprise directory service implementing LDAP (*i.e.* CA IM [1]) is used to obtain the CA IM (LDAP) dataset (henceforth referred to LDAP). We use the textual representation of LDAP interaction traces, which contain 5000 interactions of eight different request types of LDAP messages: `Add, Delete, Bind, Search, Modify DN, Modify, Compare,` and `Unbind`. Table 4.10 shows the number of interactions for each type of request messages, reflecting typical real world usage.

Table 4.10: LDAP Messages

Request Type	Bind	Add	Delete	Search	Modify	Modify DN	Compare	Unbind	Total
No. of Messages	100	928	870	1015	967	90	930	100	5000

- **BANK (SOAP):** Simple Object Access Protocol (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment [111], used by Web Services. SOAP messages use XML technologies to define an extensible messaging framework, which provides a message construct that can be exchanged over a variety of underlying protocols. The BANK (SOAP) (henceforth referred to SOAP) trace is obtained using a Banking web service and contains 5000 interactions of five different types: `createNewAccount, deposit, withdraw, getAccount` and `closeAccount.` Table 4.11 shows the number of interactions for each type of request messages.

Table 4.11: SOAP Messages

Request Type	CreateNewAccount	getAccount	deposit	withdraw	closeAccount	Total
No. of Messages	593	1383	1416	1433	175	5000

- **Twitter (REST):** Representational State Transfer (REST) is an architecture for managing information and resources using simple HTTP invocations, used by REST Web Services. The Twitter (REST) API [112] provides Web application developers with a number of services to enable automation of Twitter functionality. The Twitter (REST) (henceforth referred to Twitter) dataset used in our experiments contains 1465 interactions of six different types: `StatusesShow`, `StatusesUpdate`, `statusesuser_timelinejsonuser_id`, `statusesuser_timeline-`

`jsonscreen_name`, `Search Tweets`, and `FriendshipsShow`. Table 4.12 shows the number of interactions for each different type of request messages.

Table 4.12: Twitter Messages

Request Type	Statuses-Show	Statuses-Update	statusesuser_timelinej-sonuser_id	statusesuser_timelinej-sonscreen_name	Search-Tweets	Friendships-Show	Total
No. of Messages	258	93	33	34	626	421	1465

- **GoogleBooks (REST):** GoogleBooks (REST) (henceforth referred to GoogleBooks) is a service from Google Inc. to search full text of books and retrieve book information [113]. The GoogleBooks dataset contains 1913 interactions of requests that perform searching in a volume, retrieving a specific volume and retrieving information about public bookshelves. Table 4.13 shows the number of interactions for each type of request messages.

Table 4.13: GoogleBooks Messages

Request Type	search_volume	search_bookshelf	Total
No. of Messages	1416	497	1913

For each service, we have recorded the communications (*i.e.* the TCP/IP packets) between a client and the target service using Wireshark [114]. For each service, we sent different types of requests to the actual service provider and recorded the responses.

4.2.2 Evaluation Metrics

For a given interaction trace of a service, we use three standard evaluation metrics, that is, Precision, Recall and F-measure, to quantitatively evaluate and compare the effectiveness of our approach to *format extraction* for the request messages. As the nature of the message clustering is completely different compared to format extraction and V-measure [115] evaluates clustering more effectively compared to F-measure by solving the "problem of matching" [115], we use the V-measure for evaluating our *interaction clustering* approach.

Interaction Clustering: Two criteria are used to compute V-measure: i) *homogeneity* is to identify whether all of its clusters contain only members of a single class and ii) *completeness* is to identify where all the members of a given class are elements of the same cluster. Let N be the number of data points, C the set of classes, K the number of clusters and $A = \{a_{ij}\}$ the contingency table where a_{ij} is the number of data points that are memebrs of class c_i and elements of cluster k_j. Then, we compute the homogeneity and completeness using the following equations [115]

$$homogeneity = \begin{cases} 1, & \text{if } H(C,K) = 0 \\ 1 - \frac{H(C|K)}{H(C)}, & \text{else} \end{cases} \tag{4.5}$$

$$completeness = \begin{cases} 1, & \text{if } H(K,C) = 0 \\ 1 - \frac{H(K|C)}{H(K)}, & \text{else} \end{cases} \tag{4.6}$$

where

$$H(C|K) = -\sum_{k=1}^{|K|}\sum_{c=1}^{|C|}\frac{a_{ck}}{N} log \frac{a_{ck}}{\sum_{c=1}^{|C|} a_{ck}}$$

$$H(C) = -\sum_{c=1}^{|C|}\frac{\sum_{k=1}^{|K|} a_{ck}}{n} log \frac{\sum_{k=1}^{|K|} a_{ck}}{n}$$

$$H(K|C) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{a_{ck}}{N} log \frac{a_{ck}}{\sum_{k=1}^{|K|} a_{ck}}$$

$$H(K) = -\sum_{k=1}^{|K|}\frac{\sum_{c=1}^{|C|} a_{ck}}{n} log \frac{\sum_{c=1}^{|C|} a_{ck}}{n}$$

Based on the above calculations of homogeneity and completeness, we calculate the V-measure using the following equation

$$V\text{-}measure = 2 \times \frac{homogeneity \times completeness}{homogeneity + completeness} \tag{4.7}$$

Format Extraction: We have applied 10-fold cross-validation [109] in evaluating format extraction. For a particular request type, *true positive* is the number of request messages accepted by the inferred format of the corresponding request type, *false positive* is the number of request messages of other types that are accepted by the inferred format of that type, and *false negative* is the number of request messages of the type concerned that are rejected by the inferred format of the corresponding request type. We also calculate the average number of formats inferred for each ground-truth format. We calculate the precision, recall and the average number of formats (N) for each cluster using the following equations as described in [116]

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{4.8}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{4.9}$$

$$N = \frac{\sum_{i=1}^{|C|}(number\ of\ inferred\ formats}{\sum_{i=1}^{|C|}(number\ of\ ground\text{-}truth\ formats} \tag{4.10}$$

where C is the number of request type-based clusters.

As described in Section 4.2.1, each evaluation dataset contains different types of messages. As the number of messages per request-type are uneven, we compute the weighted score for both precision and recall using the following equations

$$Precision_w = \frac{\sum_{k=1}^{|K|}(n_k \times p_k)}{N} \tag{4.11}$$

$$Recall_w = \frac{\sum_{k=1}^{|K|}(n_k \times r_k)}{N} \tag{4.12}$$

where K is the number of clusters/request-types, N is the total number of messages, n_k is the number of messages of request type k, p_k is the precision of the inferred format for request type k and r_k is the recall of the inferred format for request type k. Finally, F-measure is calculated as the harmonic mean of precision and recall using the following equation

$$F\text{-}measure = 2 \times \frac{Precision_w \times Recall_w}{Precision_w + Recall_w} \tag{4.13}$$

4.2.3 Results

In this section, we present the experimental results (*i.e.* accuracy and efficiency) and use ProDecoder [74] and AutoReEngine [73] as the baseline to compare the accuracy of our approach as they all aim to extract type-specific message formats. The accuracy corresponding to the *interaction clustering* and *format extraction* and the efficiency of our approach are presented in the following subsections.

Interaction Clustering

Table 4.14 reports the interaction clustering results. It shows that our approach achieves very high homogeneity, completeness and V-measure in interaction clustering based on the request type for all four data sets. This is because the examined datasets contain request type field in the request messages and our approach accurately identifies the request type field through entropy analysis and considering the dissimilarity among the messages in a cluster. AutoReEngine identifies some false keywords (extracting payload information as keywords) and uses these keywords for clustering, it splits request messages of the same request type into multiple clusters and, consequently, generates more clusters than there are request message types. For instance, our results show that AutoReEngine splits LDAP `Add` request messages into 83 different clusters, compared to one cluster in our approach. In total, it generates 144 clusters for all the LDAP request messages, whereas there are only 8 different request types. ProDecoder also clusters request messages based on keywords, but it requires the number of clusters as an input parameter. Consequently, ProDecoder achieves better completeness compared to AutoReEngine. For instance, for LDAP the completeness of ProDecoder is 0.71 compared to 0.42 for AutoReEngine. However, ProDecoder fails to create clusters with low request message counts. For instance, it splits the LDAP `Search` messages into three different clusters and mixed `Bind`, `Unbind`, `Modify`, `ModifyDN` and `Delete` messages (with low request type-specific message counts) into a single cluster and hence, achieves lower homogeneity (0.67) compared to the homogeneity (0.99) achieves by AutoReEngine as AutoReEngine does not combine messages from different types into a single cluster except that a few `Delete` messages are mixed with `Compare` messages.

Similarly, for the rest of the datasets, ProDecoder mixed messages from multiple clusters into a single cluster and hence, achieves lower homogeneity and completeness. On the other hand, AutoReEngine achieves better accuracy than ProDecoder, especially completeness for SOAP, Twitter and GoogleBooks. For SOAP, AutoReEngine classifies messages accurately except that it mixes the messages of

Table 4.14: Result of Clustering Interactions Based on the Request Type

Data Set	ProDecoder			AutoReEngine			Our Approach		
	H	C	V	H	C	V	H	C	V
LDAP	0.67	0.71	0.69	0.99	0.42	0.59	1.00	1.00	1.00
SOAP	0.25	0.31	0.28	0.94	1.00	0.97	1.00	1.00	1.00
Twitter	0.79	0.66	0.72	0.74	0.92	0.82	1.00	1.00	1.00
GoogleBooks	0.62	0.64	0.63	0.00	1.00	0.00	1.00	1.00	1.00

*Note: H is homogeneity, C is completeness and V is V-measure.

request-types: `CreateNewAccount` and `closeAccount`. In the same way, it mixes the messages of different types into a single cluster for Twitter and GoogleBooks. AutoReEngines blends all messages into a single cluster instead of the actual 2 clusters, and hence achieves the homogeneity 0.00 for GoogleBooks.

However, neither AutoReEngine nor ProDecoder were able to accurately cluster the request messages according to their request types. Our approach, on the other hand, accurately identifies the type of each request message by analyzing the entropy of the message fields and considering the dissimilarity among messages, and hence correctly clustered the interaction according to the request type, outperforming both AutoReEngine and ProDecoder, achieves 100% accuracy for the 4 datasets examined.

Format Extraction

Table 4.15 reports the results of the request format extraction for the four different datasets, where N indicates the average number of formats inferred per ground-truth request-type (A more detailed result of format extraction contains the accuracy of the inferred format for each request-type based cluster is presented in Appendix A.1). As all approaches infer request formats by identifying common portions across the request messages of a cluster and different types of request messages contain different set of keywords, their accuracy in request *format extraction* highly depends on the results of the *interaction clustering*. Table 4.14 shows that AutoReEngine achieves comparatively low completeness in clustering interactions, and thus, it achieves low recall in message format inference as well (*cf.* Table 4.15). AutoReEngine overclassifies the request messages and infers request formats with some payload information being considered as keywords. For the LDAP data set,

Table 4.15: Request Format Extraction Result

Data Set	ProDecoder				AutoReEngine				Our Approach			
	P	R	F	N	P	R	F	N	P	R	F	N
LDAP	0.76	0.83	0.79	1	0.63	0.21	0.31	18	1.00	1.00	1.00	1
SOAP	0.26	1.00	0.41	1	0.97	1.00	0.99	0.80	1.00	1.00	1.00	1
Twitter	0.77	0.77	0.77	1	0.78	0.91	0.84	0.83	1.00	1.00	1.00	1
GoogleBooks	0.89	0.98	0.93	1	0.74	1.00	0.85	0.50	1.00	1.00	1.00	1

*Note: P is precision, R is recall, F is F-measure and N is the average number of formats per ground-truth format.

it generates 18 clusters on average per ground-truth cluster, *i.e.*, 144 clusters in total, including 83 clusters for LDAP `Add` request messages alone. As a consequence, the inferred request formats (one format corresponding to each cluster) become too specific and, therefore, cannot match request messages of the same type with different payload, resulting in very low recall values. Moreover, it creates an individual cluster for each `ModifyDN` messages (most of the time), which leads to achieve 0.00 in both precision and recall. It mixes the messages of `Delete` and `Compare` clusters. Therefore, AutoReEngine achieves very low accuracy in extracting formats for LDAP. On the other hand, the precision and recall of ProDecoder for LDAP are better compared to AutoReEngine. This is mainly due to the fact that the number of clusters is an input parameter and therefore, ProDecoder does not overclassify the request messages. But, ProDecoder includes request messages of different types into a single cluster and thus has worse results than our approach.

However, AutoReEngine achieves better precision and recall in extracting format for other datasets (*i.e.*, SOAP, Twitter and GoogleBooks) compared to ProDecoder as it classifies messages more accurately than ProDecoder (*cf.* Table 4.14). In SOAP, AutoReEngine achieves 100% precision and recall in extracting format for all clusters except for `CreateNewAccount` as it creates mixed cluster and thus, achieves 0.97 as weighted precision. In contrast, ProDecoder achieves lower homogeneity and completeness in clustering but, it achieves higher recall in format extraction. This is so because, even though ProDecoder mixed messages of multiple clusters into a single cluster, it infers the format through identifying the common message sequences among messages in a cluster and therefore, it does not accept the message of other types especially when the mixing is highly asymmetrical. For example, ProDecorer constitutes the `deposit` cluster with 1120 messages of `deposit` type and

137 messages of `CreateNewAccount` type and the inferred format for the `deposit` cluster contains keywords (most) of `deposit` messages and hence, does not accept messages of other types, *i.e.*, achieve 1.00 recall. In the same way, it achieves comparatively better precision and recall in format extraction for Twitter and GoogleBooks datasets than in clustering. On the other hand, AutoReEngine achieves lower precision and recall in extracting format for Twitter compared to SOAP and GoogleBooks and to Twitter messages clustering. The reason is that AutoReEngine creates 4 clusters instead of 6 clusters and mixed the messages of `statusesupdate` and `statusesuser_timelinejsonuser_id` clusters into a single cluster and achieves 0.00 precision and recall for the inferred format for that cluster. Similarly, it achieves low precision and recall for the type `friendshipsshow` as the cluster contains the messages of `statusesshow` type, therefore, infers imprecise format for that cluster and achieves relatively lower precision and recall in format extraction for Twitter dataset.

In general, our approach not only achieves high accuracy in interaction clustering, but also achieves very high accuracy in request format inference. Moreover, our approach infers request formats with generalization in the form of regular expression. Thus, it can accept request messages with unseen, yet valid patterns. For example, LDAP `Add` request messages may contain organizational units multiple times. In the format inferred by ProDecoder and AutoReEngine, '`ou`' appears exactly two times, but it could be repeated more than two times (as in the dataset). In contrast, the format inferred by our approach accepts multiple occurrences of '`,ou=`' in LDAP `Add` request messages (*cf.* the regular expression given at the end of Section 4.1).

Efficiency

While achieving high accuracy in message format extraction is our main objective, we also quantify the impact on the efficiency of our approach. We instrument the code of clustering the messages and inferring formats of the messages and record the times for all datasets. We run the experiments on a machine with Intel Core(TM) i5-4570 3.20 GHz with 16GB of main memory. For comparison we run our approach and the compared techniques (*i.e.*, ProDecoder and AutoReEngine) on the same machine and logged the required time of clustering and extracting formats.

Table 4.16 shows the average[2] time (seconds) for clustering the interactions based on the request type and inferring the formats from the request messages for each of the request-type based clusters. As Table 4.16 shows, our approach requires

[2]Each approach infers a set of 10 formats per request-type based cluster as the 10-fold cross-validation technique is used to evaluate the result.

Table 4.16: Average Time (in seconds) of Inferring Request Format

Data Set	ProDecoder		AutoReEngine		Our Approach	
	T_C	T_F	T_C	T_F	T_C	T_F
LDAP	246.95	39.68	4.80	0.54	2571.19	3.81
SOAP	437.69	250.03	58.36	159.29	3414.54	81.62
Twitter	63.20	2.92	3.24	6.40	82.32	1.64
GoogleBooks	56.11	13.00	2.31	23.29	56.28	1.85

*Note: T_C is the required time (s) for clustering and T_F is the required time (s) for inferring formats.

a comparatively longer time in clustering the interactions based on the request type. This is so because of the adopted template generation technique [11] utilizes multiple sequence alignment (MSA) [107] for aligning the request messages and it requires a long time to get the alignment result, especially for the lengthy messages. However, our approach requires less time in request format inference compared to ProDecoder and AutoReEngine except for AutoReEngine over LDAP. The reason is that the request messages for the LDAP dataset are relatively longer than the request messages of other datasets and thus, our approach generates more n-grams from the LDAP request messages as candidate keywords in the keyword extraction step, which requires more computation time. Nevertheless, the accuracy of the format inference is much important than the efficiency, and in particular, the format inference is done in the offline phase before the inferred formats are used in response generation (online) phase.

4.3 Discussion

In this section, we further reflect on our approach and discuss a few assumptions and limitations our approach has.

The first point to note that we assume that the interaction traces are not encrypted. Otherwise, it would normally be impossible to analyze the message structure. Furthermore, we focus on services with textual messaging protocols, that is, the service interactions are in a textual form even though the particular protocol followed by the service provider does not need to be known. This is because the keywords in binary protocols may be less than one byte, where the identification of keywords and request type fields requires different techniques. However, our ap-

proach may still work with binary messaging protocols with the atomic message fields of one byte or longer (*i.e.*, no Bit fields). Furthermore, we have assumed that the messages have a request type field. We consider this assumption as realistic because we have not encountered messages of a service that does not have a type field.

In general, the request formats extracted using our approach depend on the given interaction trace rather than the corresponding service. That is, it can only identify the formats for those types of request messages that are present in the interaction trace. Therefore, the diversity of the request types and variations is important to the extracted request types' coverage for the given service in its message trace. In particular, if there is only one message instance for a given request type, the whole message will be identified as a keyword for that request type due to the fact that there is no variation identifiable between the request messages of the same type. As such, it is important for the interaction trace to contain diverse request types, with variations and instances, as much as possible.

Moreover, our approach selects a number of message fields as the candidate request type fields based on the entropy of the fields. It utilizes the observation that the request type field has the *least* data variation among all the variable fields. But, there may be service protocols where certain message fields may have even less variability than the request type field. For example, some messages in a particular protocol may have a binary field with `true` or `false` as possible values. Such message field will be selected as candidate message type field. Finally, the request type field is selected by considering that the most accurate clustering has the lowest dissimilarity among the request messages in a cluster. This consideration ensures the messages in the same cluster have the lowest dissimilarity among themselves, while the messages in separate clusters have the highest dissimilarity among themselves. This suggests that our approach can accurately cluster the messages as long as the messages of a single cluster have the lowest dissimilarity even if an inaccurate message field is selected by the entropy-based technique in the preceding step.

Another point of interest concerns the difference between a service in general and a particular deployment thereof. For example, for the LDAP interaction trace, we have identified '`ca.com`' as part of a keyword. From the overall service viewpoint, it is part of payload (email address), not part of a keyword. From a particular deployment viewpoint, however, it is important and useful for it to be identified as part of a keyword because the particular Identity Manager (IM) [1] is deployed at CA and all email addresses should contain '`ca.com`'; otherwise, it may indicate a breach of the organizational protocol (*i.e.*, non-CA emails are recorded in the CA directory).

On the other hand, if the service deployment serves multiple organizations or the organization does not dictate the use of official email, such a string will not be part of a keyword (due to variation between request messages). Similarly, if the data set contains interaction traces from different deployments, such deployment-specific payload will not appear as part of a keyword in the extracted formats.

4.4 Summary

In this chapter, we have presented an approach to extracting fine-grained request type-specific formats for a given service from its interaction traces. Our approach does not require any prior knowledge about the service or its message structure, nor does it need access to the source code. In our approach, we first identify the request type field through message alignment and entropy analysis, and use the request type field to partition the interactions (messages) into request type-specific clusters. Then, we identify the keywords for each message cluster by breaking the request messages down to words and analyzing the frequency of their independent occurrences. Finally, we use the keywords to tokenize the request messages, and derive the request formats by analyzing the order of appearance of the message tokens (keywords and payload segments). We have been able to achieve high accuracy in clustering the interactions based on the request type and in extracting formats of the request messages, showing greater improvements over the state of art approaches. This has been demonstrated through a range of comparative experiments using datasets from real-world services that follow different messaging protocols.

We will present our approach of inferring formats for response messages in the next chapter as the response format inference has different challenges than extracting request formats.

Chapter 5

Format Extraction for Response Messages

In the previous chapter, we have presented our approach to extracting the formats of the type-specific request messages by identifying the request type field from the request messages. The interactions are clustered based on the request type, including all of the response messages for the request messages concerned. In this chapter, we present our approach to fulfilling another key requirement, *i.e.*, inferring the formats of the response messages to separate the payloads from the message keywords. We cluster the response messages of each request-based cluster based on the response type. Then, the format of the response messages for each response-based cluster is inferred by extracting the type-specific keywords from the response messages of that cluster. As explained in Section 2.3.2, a number of issues different from inferring request formats need to be addressed for inferring formats for the response messages.

The rest of the chapter is organized as follows. In Section 5.1, we present the result of clustering the interactions based on the request type. In Section 5.2, we present the details of our approach to inferring the formats for the response messages. The experimental results of applying our approach to the traces collected from a number of real services are presented in Section 5.3. Finally, we summarize this chapter in Section 5.4.

5.1 Preliminaries

As described in Section 2.3.2, a service generates different types of responses for the different types of requests and even for the same type of requests. Each type

of response message has its own set of keywords and its own format. Thus, the extraction of the formats of the response messages requires to cluster the interactions based on the request type. This section presents the result of clustering the interaction trace (*cf.* Table 4.1) based on the request type. We use the request type identification technique [105] as described in Chapter 4 for identifying the request type field from the request messages. It clusters the example interactions in Table 4.1 into four groups: **`Bind, Add, Search`** and **`Unbind`**. Table 5.1 shows the responses of the **`Search`** cluster after clustering the interactions of Table 4.1 based on the request-type. We will use this example response cluster to illustrate our approach in the next section.

Table 5.1: Response Messages of **`Search`** cluster

No.	Responses
Resp#5	LDAP Search Result Done Message ID: 5 LDAP Search Result Done Protocol Op Result Code: 32 (No Such Object) Matched DN: ou=Customer,dc=ca,dc=com
Resp#6	LDAP Search Result Entry Message ID: 6 LDAP Search Result Entry Protocol Op dn: cn=Clive BRANCH,ou=Finance,ou=Corporate,dc=ca,dc=com cn: Clive BRANCH mail: Clive.BRANCH@ca.com mobile: 6312753 description: Design Administrator objectClass: inetOrgPerson title: Financial Economist sn: BRANCH LDAP Search Result Done Message ID: 6 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
Resp#7	LDAP Search Result Entry Message ID: 7 LDAP Search Result Entry Protocol Op dn: cn=Eddy BRYCE,ou=Construction,ou=Projects,dc=ca,dc=com cn: Eddy BRYCE mail: Eddy.BRYCE@ca.com mobile: 5940538 description: Software Consultant objectClass: inetOrgPerson title: Communications Services Co-ordinator sn: BRYCE LDAP Search Result Entry Message ID: 7 LDAP Search Result Entry Protocol Op dn: cn=Gwen HUNTER,ou=Construction,ou=Projects,dc=ca,dc=com cn: Gwen HUNTER mail: Gwen.HUNTER@ca.com mobile: 6340642 description: Response Engineer objectClass: inetOrgPerson title: Purchasing Consultant sn: HUNTER LDAP Search Result Done Message ID: 7 LDAP Search Result Done Protocol Op Result Code: 0 (Success)

Table 5.1 Continued: Response Messages of **Search** cluster

No.	Responses
Resp#8	LDAP Search Result Entry Message ID: 8 LDAP Search Result Entry Protocol Op dn: cn=William SIMPER,ou=Training,ou=Human Resources,dc=ca,dc=com cn: William SIMPER mail: William.SIMPER@ca.com mobile: 6813842 description: Computing Officer objectClass: inetOrgPerson title: Consulting Technician sn: SIMPER LDAP Search Result Entry Message ID: 8 LDAP Search Result Entry Protocol Op dn: cn=Joseph GRIMES,ou=Training,ou=Human Resources,dc=ca,dc=com cn:Joseph GRIMES mail: Joseph.GRIMES@ca.com mobile: 6953740 description: Training Officer objectClass: inetOrgPerson title: Industrial Clerk sn: GRIMES LDAP Search Result Done Message ID: 8 LDAP Search Result Done Protocol Op Result Code: 0 (Success)
Resp#9	LDAP Search Result Entry Message ID: 9 LDAP Search Result Entry Protocol Op dn: cn=Brad DUFFY,ou=Industrial Relations,ou=Customer,dc=ca,dc=com cn: Brad DUFFY mail: Brad.DUFFY@ca.com mobile: 8219206 description: Hardware Support objectClass: inetOrgPerson title: Acting Engineer sn: DUFFY LDAP Search Result Done Message ID: 9 LDAP Search Result Done Protocol Op Result Code: 0 (Success)

5.2 Approach

In this section, we present our approach that takes the response messages of the request-based cluster as input and automatically infers the formats of the response messages. The key to achieving this goal is the ability to (i) accurately further cluster the responses for each request-based cluster, (ii) accurately identify the keywords from the response messages (payload overwhelm messages) by considering the number of messages that contain a keyword by removing the sub-strings of a keyword using frequency subtraction, and (iii) generalize the individual message patterns into the inferred format to handle the pattern with *repetition* in the response messages. Figure 5.1 shows the steps of our approach. We describe each of the components after presenting an overview of our approach.

Our approach has two major steps: i) further clustering the responses of each request-based cluster according to the response messages, and iii) inferring the formats for each response-based cluster by identifying the cluster-level keywords from

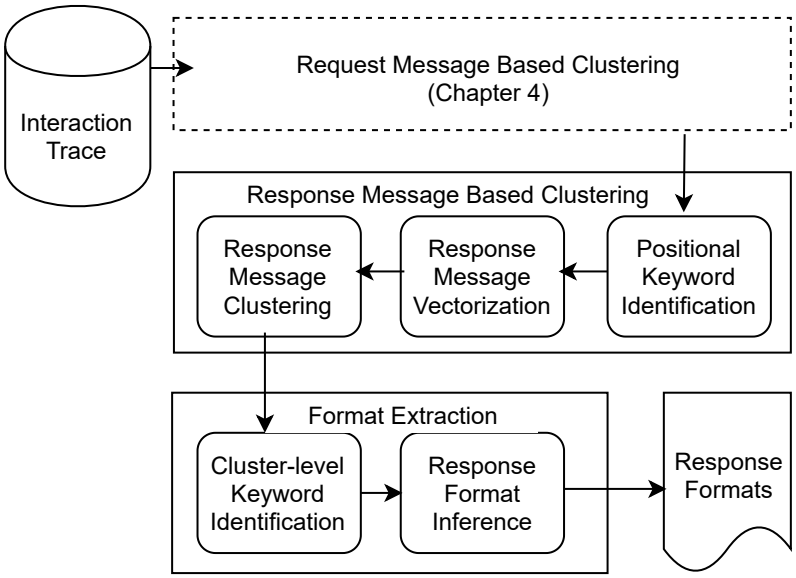


Figure 5.1: An Overview of Response Format Extraction Approach

the response messages in that cluster. In the first step (Response Messages Clustering), we group the response messages of a request-based cluster into clusters based on the response messages by identifying the co-occurrence of the positional keywords from the response messages. We adopt our recently proposed clustering technique to cluster the response messages [106]. We use a different technique than the request message clustering to cluster the responses because the response messages do not always contain a message type field as in the request messages. Moreover, the identification of fixed portions across the response messages using the common template generation technique [11] is not applicable as the different types of responses have a different set of message keywords and usually, they do not share the similar keywords. Thus, to cluster the responses, we break the response messages into a set of tokens. Then, only high-frequency tokens and their positions in the response messages are considered as candidate positional keywords from the tokenized response messages. The same keywords appearing in nearby positions are merged based on a position window and finally, the response messages are clustered based on the co-occurrence of positional keywords.

In the second step (Format Extraction), every response message in a cluster is converted into a sequence of alternating keyword and payload using the extracted keywords from the response messages of the respective cluster. Finally, we infer the format in the form of a Finite State Machine (FSM) for each response-based cluster and convert the FSM to a regular expression as the response format for the cluster.

Figure 5.2 shows the clustering result of the interactions listed in the Table 4.1. After clustering the responses, we extract a set of keywords from the response messages for each response based cluster. Finally, the format of the response messages

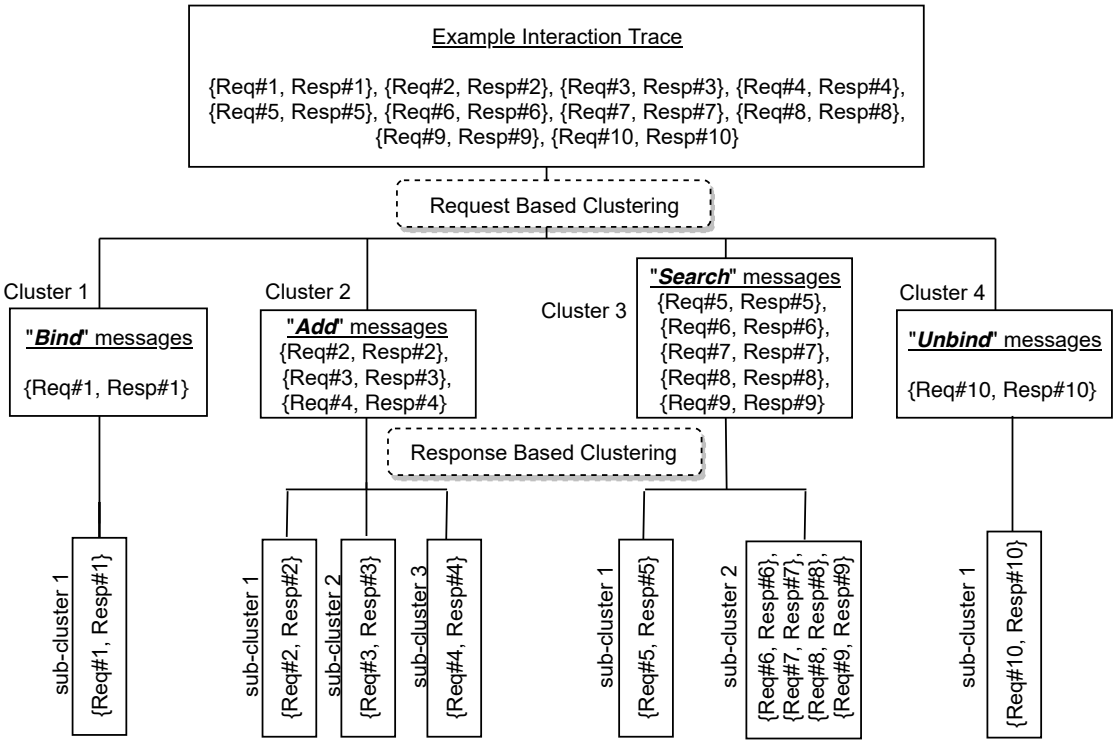


Figure 5.2: Clustering Result of LDAP Example Interaction Trace

are inferred for each response based cluster, *i.e.*, for the response messages of each sub-cluster in Figure 5.2.

5.2.1 Response Messages Clustering

The purpose of this step is to cluster the response messages for each request-based cluster. A service usually supports different types of requests and even generates different types of responses for the same type of requests at different times. Therefore, different types of responses need to be identified in order to extract the message formats. Moreover, each type of responses generated for a particular request type has its own set of keywords and has its own format. Thus, the responses in the request-based cluster need to be clustered further based on the response types to accurately extract keywords and, hence, infer precise format from the response messages. For example, in Table 5.1, the keywords and the format of the generated responses `Resp#5` and `Resp#6` are different. But, we use a different technique to cluster the responses because of two reasons: (i) response messages may not have the message type field and (ii) it is not applicable to generate a common template and identify the fixed and variable portions of the response messages.

For example, the identification of fixed and variable portion for the responses of the `Search` cluster (*cf.* Table 5.1) by generating a common template is not applica-

ble as the position of the keywords (*i.e.*, `LDAP Search Result Done Message ID:`) is different in different response messages even though they share the keywords across different types of response messages. Let us consider the GoogleBooks interactions as another example. As Table 5.2 shows, both requests retrieve information about a bookshelf (for a given bookshelf id), where the *bookshelf id* for the first request is "`499`" and the second request is "`0`". The server returns an error response for the first request as it unable to find a bookshelf with id "`499`", whereas the server returns the detailed information about a bookshelf (*i.e.*, the title of the bookshelf, the date of creation and modification, the number of volumes, etc.). It indicates that the same request with different *bookshelf id* returns different types of responses and the responses contain a different set of message keywords. But, none of the message fields (common in both responses) defines the response type. Moreover, the common template generation technique (*cf.* Chapter 4) is not applicable in separating the fixed and variable portions of the response messages as they do not have the common message keywords. As such, we use our recently proposed clustering technique, P-token[106] to cluster the responses. It effectively clusters the messages based on the positional keywords. It has three sub-steps: (1) identifying the positional keywords from the response messages, (2) merging keywords based on position windows and (3) vectorizing response messages and cluster responses based on the co-occurrence of the positional keywords.

Table 5.2: GoogleBooks Interactions (`bookshelves`)

No.		Interactions
1	Req	`https://www.googleapis.com/books/v1/users/107782646712117400162/bookshelves/499?key=**********`
	Resp	`{ "error": { "errors": [{ "domain": "global", "reason": "notFound", "message": "The bookshelf ID could not be found.", "locationType": "other", "location": "backend_flow" }], "code": 404, "message": "The bookshelf ID could not be found." }}`
2	Req	`https://www.googleapis.com/books/v1/users/107782646712117400162/bookshelves/0?key=**********`
	Resp	`{ "kind": "books#bookshelf", "id": 0, "selfLink": "https://www.googleapis.com/books/v1/users/107782646712117400162/bookshelves/0", "title": "Favorites", "access": "PUBLIC", "updated": "2007-11-10T18:02:25.555Z", "created": 2007-11-10T18:02:25.555Z", "volumeCount": 6, "volumesLastUpdated": "2019-10-01T12:56:51.000Z"}`

Identifying Positional Keywords

In this sub-step, we identify keywords from the response messages based on the following criteria: keywords are common to the same type of response messages and they appear at relatively fixed positions across the response messages. This sub-step involves (i) extracting the candidate positional keywords through frequency analysis of tokens, and (ii) identifying the true keywords through studying the occurrence and recurrences of candidate positional keywords. First, a response message in a given trace (*i.e.*, request-based cluster) is split into tokens using tokenization [117], and the position of each token is recorded. For example, the first response message (`Resp#5`) in Table 5.1 can be broken down to tokens$_{(position)}$: "`LDAP`$_{(0)}$", "`Search`$_{(1)}$", "`Result`$_{(2)}$", "`Done`$_{(3)}$", "`Message`$_{(4)}$", "`ID`$_{(5)}$", etc. Then, the *candidate positional keywords* are extracted based on the number of response messages in which a token appears, rather than the number of all occurrences of a token in all messages. This consideration emphasizes the importance of a token's appearance across different messages, and de-emphasizes the multiple occurrences of a token in a single message, because a keyword we are looking for should appears in all messages of the same type. A threshold has been applied to select only high frequency keywords as the candidate positional keywords. For example, Table 5.3 shows some positional candidate keywords extracted from the response messages of `Search` cluster (*i.e.*, from `Resp#5`, `Resp#6`, `Resp#7`, `Resp#8` and `Resp#9`) with the threshold `2` (*i.e.*, keyword appears at least in 2 messages).

Table 5.3: Candidate Positional Keywords

Candidate Keyword	Position	Frequency
`Message`	4	5
`Message`	71	2
`Entry`	11	4
`Entry`	3	4
`mail`	40	2
`mail`	41	2

Merging Keywords

In this step, we merge the positional keywords in the same "window", *i.e.*, those appearing at slightly different positions in different messages. For example, the same keyword "`mail`" in Table 5.3 extracted twice as it appears in two different positions (*i.e.*, 40 and 41) in the messages because of variable length of payloads. We merge

the keywords fall into the same window and use "$\texttt{mail}_{(40)}$" (at the lowest position) to represent the two positional tokens, update frequency $f(\texttt{mail}_{(40)})$ to 4, and remove "$\texttt{mail}_{(41)}$" from the candidate positional keywords as they correspond to the same positional keyword. We compute the window size based on the standard deviation of the keyword's position for each keyword and adopts the Parzen-Window Density Estimation [118] to estimate the window size.

$$\delta(t) = 1.06 \cdot \sigma_t \cdot V_t^{-1/5}, \tag{5.1}$$

where σ_t is the standard deviation of t's (*i.e.*, token) positions and V is the volume (variation) of t's positions. It estimates the window sizes to be 27.93, 3.69 and 3.57 for the keywords "**`Message`**", "**`Entry`**" and "**`mail`**" respectively. So, the two "**`mail`**" keywords in Table 5.3 are merged together and update its frequency to 4, while the positional keywords of "**`Message`**" and "**`Entry`**" are kept as separate keywords. For example, Table 5.4 shows the positional keywords after merging the positional keywords in Table 5.3.

Table 5.4: Positional Keywords after Merging

Keyword	Position	Frequency
`Message`	4	5
`Message`	71	2
`Entry`	11	4
`Entry`	3	4
`mail`	40	4

Vectorizing and Clustering Response Messages

In this sub-step, we group the response messages into clusters so that each cluster is of high homogeneity in terms of the positional keyword sequence across all the response messages in the cluster. This sub-step involves (i) vectorizing the response messages based on the positional keywords, and (ii) clustering the response messages using VAT [119]. We use the positional keywords extracted from the last sub-step as *features* in VAT to cluster the response messages for each request-type based cluster. We use the set S to denote the extracted features. In total, we have $|S|$ features, where $|S|$ denotes the cardinality of set S. Hence, for an arbitrary message m, we can define a $1 \times |S|$ vector, v_m, as follows:

$$v_m = [w(t_{(i)})]_{1\times|S|},\ \forall\ t_{(i)} \in S, \tag{5.2}$$

where, $w(t_{(i)})$ is the weight of feature $t_{(i)}$ (positional keyword of token t at position i) in message m. For each feature $t_{(i)} \in S$, we measure its weight in message m by examining if it is covered by the token sub-sequence $\{t^m_{(i)}, t^m_{(i+1)} \cdots t^m_{(i+\delta(t_{(i)}))}\}$ in message m, where $t^m_{(j)}$ denotes the j-th token in message m and $\delta(t_{(i)})$ is the window size of the inspected feature $t_{(i)}$. If $t_{(i)}$ is not covered by the sub-sequence, we set $w(t_{(i)}) = 0$; otherwise, set $w(t_{(i)}) = 1$. For example, we put '1' in k_2 column for `Resp#5` in the following vector as the response contains the keyword **Message**$_{(4)}$, where k_1, k_2, k_3, k_4 and k_5 correspond to the keywords **Entry**$_{(3)}$, **Message**$_{(4)}$, **Entry**$_{(11)}$, **mail**$_{(40)}$ and **Message**$_{(71)}$ respectively.

$$v_m = \begin{array}{c} \\ Resp_5 \\ Resp_6 \\ Resp_7 \\ Resp_8 \\ Resp_9 \end{array} \begin{array}{c} \begin{array}{ccccccc} ... & k_1 & k_2 & k_3 & k_4 & k_5 & ... \end{array} \\ \begin{pmatrix} & 0 & 1 & 0 & 0 & 0 & \\ & 1 & 1 & 1 & 1 & 0 & \\ & 1 & 1 & 1 & 1 & 0 & \\ & 1 & 1 & 1 & 1 & 1 & \\ & 1 & 1 & 1 & 1 & 1 & \end{pmatrix} \end{array} \tag{5.3}$$

Then, we convert each message into a weighted vector based on the positional keywords appearing in the message. Finally, we adopt Needleman-Wunsch [15] to calculate the dissimilarity between response messages and VAT [119] to cluster the response messages with the aim to have the lowest dissimilarity among messages in a cluster. For example, it generates the following dissimilarity matrix for the response messages of the **Search** cluster (*i.e.*, **Resp#5**, **Resp#6**, **Resp#7**, **Resp#8** and **Resp#9**). As the dissimilarity matrix (*cf.* eq. 5.4) shows, **Resp#5** has high dissimilarity with the rest of the response messages of the **Search** cluster and thus, the response messages are grouped into two clusters, *i.e.*, **Resp#5** in one cluster and others in another cluster.

$$D_m = \begin{array}{c} \\ Resp_5 \\ Resp_6 \\ Resp_7 \\ Resp_8 \\ Resp_9 \end{array} \begin{array}{c} \begin{array}{ccccc} Resp_5 & Resp_6 & Resp_7 & Resp_8 & Resp_9 \end{array} \\ \begin{pmatrix} 0.0 & 0.61 & 0.64 & 0.65 & 0.62 \\ 0.61 & 0.0 & 0.09 & 0.10 & 0.15 \\ 0.64 & 0.09 & 0.0 & 0.01 & 0.05 \\ 0.65 & 0.10 & 0.01 & 0.0 & 0.04 \\ 0.62 & 0.15 & 0.05 & 0.04 & 0.0 \end{pmatrix} \end{array} \tag{5.4}$$

5.2.2 Format Extraction

The purpose of this step is to infer a response format for each response-based cluster. The inferred format is presented in the form of a *regular expression*, an alternate sequence of keywords and payload information. This step has two major sub-steps: (i) Cluster-level Keyword Identification and (ii) Format Inference.

Cluster-level Keyword Identification

The purpose of this sub-step is to identify a set of keywords from the response messages of each cluster. It accepts all the response messages of the same cluster as an input and generates a set of keywords for that cluster as an output. It generates keywords through i) identifying a set of candidate keywords by analyzing the appearance of n-grams and ii) selecting the true keywords from the candidate keywords by removing the candidate keywords that are sub-string of another keywords.

Algorithm 4 shows the dedicated algorithm of generating a set of candidate keywords. At first, each response message in a cluster is split into words (or grams) of length n using the n-grams technique (lines *6* to *16*). After breaking the response messages, it creates a list of message IDs for each n-gram to count the number of response messages containing the n-gram. It emphasizes the importance of candidate keyword's appearance across response messages instead of calculating frequency (*i.e.*, *repetitive* counting) of all occurrences of candidate keywords in a message. As the response messages usually contain some *repetitive* patterns, some of the payloads in the *repetitive* patterns may appear multiple times and very likely to be picked as keywords if we count all occurrences of the n-gram. At line *6*, we loop over all the messages in a cluster M and keep track of the current message index or ID in j. At line *7*, the j-th message is broken into consecutive sequence of characters (n-grams) of length n. We set the initial value of N to be *2* for allowing it to identify all the keywords of minimum length *2*. We create a list with message index for each n-gram (lines *8* to *14*). At line *17*, the pair of n-gram and its number of occurrences is generated by counting the number of unique message indices in the list for that n-gram. A threshold is used to select the candidate keywords (lines *19* to *24*). As we apply the keyword extraction for each cluster and the messages of a cluster share a similar format, we set the threshold T to *1.00* for selecting those keywords that appear once (at least) in every messages. At line *25*, the length of the grams n is incremented by 1, and process continues for finding the candidate keywords of length *n+1*, and the whole process continues until no new candidate keywords are found. In the end, Algorithm 4 returns a set of candidate keywords

Algorithm 4: Candidate Keyword Generation

Input: minimum length N, threshold T, message set M
Initialize: keyword tuples $K(keyword, number) = L(keyword, number) = \emptyset$, message ID set D= $\emptyset$, keyword multiset $W = \emptyset$
$n \leftarrow N$
Boolean $found \leftarrow true$
while $found = true$ **do**
  for $m_j \in M$ **do**
    for $y_i \in \{x_i..x_{i+n-1} \in m_j | i = 0..length(m_j)\text{-}n\}$ **do**
      $D \leftarrow W[y_i]$
      if $D = \emptyset$ **then**
        $W \leftarrow W \cup \{y_i, D\}$
      end if
      if $j \notin D$ **then**
        $D \leftarrow D \cup \{j\}$
      end if
    end for
  end for
  $L(keyword, number) \leftarrow$ CountMessageIDs(W)
  $found \leftarrow false$
  for $k(keyword, number) \in L$ **do**
    if $number \geq T * |M|$ **then**
      $K \leftarrow K \cup \{k\}$
      $found \leftarrow true$
    end if
  end for
  $n \leftarrow n + 1$
end while
Return K

with the numbers of their appearances in the messages, *i.e.*, the number of messages containing each of them. For example, Table 5.5 shows some candidate keywords and their appearances in the messages that are extracted using Algorithm 4 from the response messages of the `Search (success)`[1] sub-cluster, *i.e.*, from the `Resp#6`, `Resp#7`, `Resp#8` and `Resp#9` in Table 5.1.

The generated candidate keyword list contains the true keywords and their sub-strings. The sub-strings of a keyword are also frequent and appear in the same number of messages. For example, all the sub-strings of a keyword "`LDAP Search Result Entry Message ID:`" appear the same number of times in the messages as the keyword appears. Thus, in this step, Algorithm 3 presented in the Chapter 4 is used to identify the true keywords from the list of candidate keywords by frequency subtraction. For example, Table 5.6 shows the candidate keywords and their corresponding

[1]The response messages of the *sub-cluster 2* of the `Search` request cluster in Figure 5.2

Table 5.5: Candidate Keywords

No.	Candidate Keywords	No. of appearance
1	`LDAP Search Result Entry Message ID:`	4
2	`LDAP Search Result Entry Message`	4
3	`LDAP Search Result Entry`	4
4	`objectclass: inetOrgPerson`	4
5	`objectclass:`	4
6	`object`	4

appearance values after subtracting substring's appearances from the number of appearances of superstring or super keyword. As Table 5.6 shows, two keywords (*i.e.*, "**`LDAP Search Result Entry Message ID:`**", "**`objectclass: inetOrgPerson`**") are extracted from the candidate keywords in Table 5.5.

Table 5.6: Keywords After Subtraction

No.	Keywords	No. of appearance
1	`LDAP Search Result Entry Message ID:`	4
2	`LDAP Search Result Entry Message`	0
3	`LDAP Search Result Entry`	0
4	`objectclass: inetOrgPerson`	4
5	`objectclass:`	0
6	`object`	0

Format Extraction

The purpose of this sub-step is to infer the message format from the response messages in each response-based cluster with the set of identified keywords in the keyword identification step. The inferred format is presented in the form of a *regular expression*, an alternate sequence of keywords and payload information. We use the same format inference technique described in Section 4.1.3 to infer the format from the response messages, which involves *tonenization* and *format inference*. For example, the following sequence is generated for the third response (*i.e.*, **`Resp#7`**) in Table 5.1 (space character is shown as '␣') after tokenization.

`LDAP␣Search␣Result␣Entry␣Message␣ID:␣`***`VARIABLE`***`␣LDAP␣ Search␣Result␣Entry␣Protocol␣Op␣dn:␣cn=`***`VARIABLE`***`,ou= ␣`***`VARIABLE`***`,ou=`***`VARIABLE`***`,dc=ca,dc=com␣cn:␣`***`VARIABLE`***`␣mail:␣`***`VARIABLE`***`@ca.com␣mobile:␣`***`VARIABLE`***`␣description: `***`VARIABLE`***`␣objectClass:␣inetOrgPerson␣title:␣`***`VARIABLE`***`␣ sn:␣`***`VARIABLE`***`␣LDAP␣Search␣Result␣Entry␣Message␣ID:␣`***`VARIABLE`***`␣LDAP␣Search␣Result␣Entry␣Protocol␣Op␣dn:␣ cn=`***`VARIABLE`***`,ou=␣`***`VARIABLE`***`,ou`

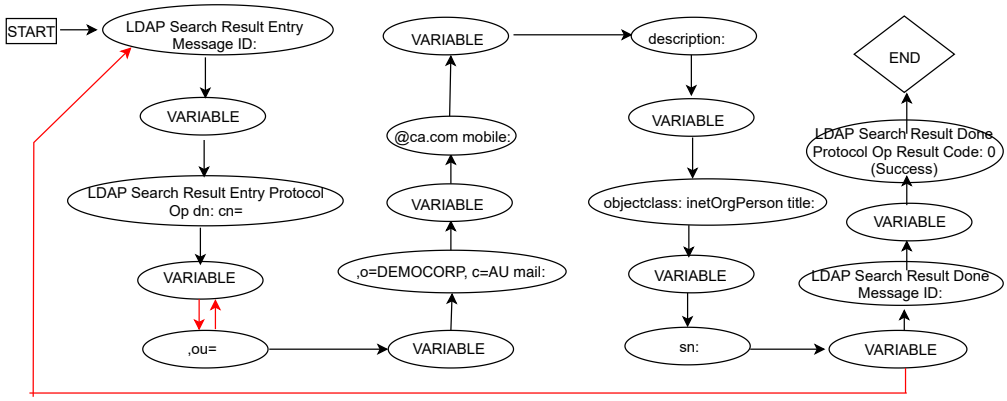


Figure 5.3: Inferred FSM by Synoptic for the **`Search Success`** response

`=`***`VARIABLE`***`,dc=ca,dc=com ␣cn:␣`***`VARIABLE`***`␣mail:␣`***`VARIABLE`***`@ca.com␣mobile:␣`***`VARIABLE`***`␣ description:`***`VARIABLE`***`␣objectClass:␣inetOrgPerson␣title: ␣`***`VARIABLE`***`␣sn:␣`***`VARIABLE`***`␣LDAP␣Search␣Result␣Done␣ Message␣ID:␣`***`VARIABLE`***`␣ LDAP␣Search␣Result␣ Done␣Protocol ␣Op␣Result␣Code:␣0␣(Success)`

Figure 5.3 shows an inferred FSM by Synoptic [76] for the responses of the **`Search (Success)`** cluster

Finally, our approach infers the following format from the response message of the **`Search (Success)`** cluster (space character is shown as '␣'). As the inferred format shows, our approach is able to generalize the *repetition* (highlighted in "red") sequences of keyword-payload allowing the inferred format to accept the valid yet unseen messages with different number of repetitions.

`(LDAP␣Search␣Result␣Entry␣Message␣ID:␣.*␣LDAP␣ Search␣Result␣Entry␣Protocol␣Op␣dn:␣cn=(.*,ou=)`$^{+}$`.*,dc=ca,dc=com␣cn:␣.*␣mail:␣.*@ca.com␣mobile:␣.*␣description:.*␣objectClass:␣inetOrgPerson␣title:␣.*␣sn:␣.*␣)`$^{+}$`␣LDAP␣ Search␣ Result␣Done␣Message␣ID:␣.*␣LDAP␣Search␣Result ␣Done␣Protocol␣Op ␣Result ␣Code:␣0␣(Success)`

5.3 Evaluation Results

We have evaluated our approach to inferring response message formats of service APIs and compared it with two state of art approaches (*i.e.*, ProDecoder [74] and AutoReEngine [73]), by applying to the interaction traces of real-world services. We run the experiments on the same datasets as described in the Chapter 4 and use the same evaluation metrics and comparative techniques to evaluate and compare the effectiveness of our approach. Again, similar to Chapter 4, we describe the experimental results in terms of accuracy and efficiency in the following sections.

5.3.1 Accuracy

In this section, we present the accuracy of clustering the responses and extracting formats for each cluster by our approach and the comparative techniques. We use ProDecoder [74] and AutoReEngine [73] as the baseline for comparison as they all aim to infer formats of the messages.

Message Clustering

Table 5.7 reports the overall result (message-volume weighted) of response message clustering, while the detailed result of clustering response messages for each request-based cluster is presented in A.2. It shows that our approach achieves 100% homogeneity, completeness, and V-measure in the message clustering step for all four data sets and outperforms the compared approaches for all the datasets.

Our approach outperforms both AutoReEngine and ProDecoder in clustering response messages as shown in Table 5.7. ProDecoder clusters messages through keyword identification based on the relationships between keywords learned by using the LDA [120], from Natural Language Processing. Unlike natural languages, the machine-generated messages follow a defined sequence of keywords and payloads (*i.e.*, message format). Hence, direct use of the technique inspired by natural language (*i.e.*, LDA) on such machine-generated messages has limitation in extracting the relationships between message keywords correctly. In contrast, our clustering approach considers the position (a machine language oriented characteristic) of the keywords in extracting message keywords from the keyword-payload formatted messages and clusters the messages based on the extracted keywords. Therefore, our approach outperforms ProDecoder in clustering response messages for all datasets. As Table 5.7 shows, ProDecoder achieves 0.69 in V-measure for LDAP, while our approach achieves 1.00. Similarly, ProDecoder also achieves lower accuracy in clustering responses for other datasets.

However, like our approach, AutoReEngine considers the position of keywords in the keyword identification step. The candidate keywords with a low standard deviation of positions are considered as keywords in AutoReEngine; otherwise they are filtered as noise. But, several false keywords (*i.e.*, payloads) are extracted from the messages, especially from the messages (*i.e.*, responses) with *repetitive* sequences of keyword-payload. Meanwhile, some keywords (true) may have large variations in their positions in the messages due to the significant variation of payloads (*i.e.*, length of payloads). As it uses such false keywords in clustering, AutoReEngine generates a lot more clusters than the actual clusters for the response messages of

Table 5.7: Clustering Result (Response Messages)

Data Set	ProDecoder			AutoReEngine			Our Approach		
	H	C	V	H	C	V	H	C	V
LDAP	0.73	0.65	0.69	0.30	0.36	0.33	1.00	1.00	1.00
SOAP	0.50	0.52	0.51	1.00	0.77	0.87	1.00	1.00	1.00
Twitter	0.62	0.65	0.63	0.82	0.11	0.20	1.00	1.00	1.00
GoogleBooks	0.29	0.27	0.28	1.00	0.31	0.48	1.00	1.00	1.00

*Note: H is homogeneity, C is completeness and V is V-measure.

having payload variations and repetitions. Therefore, it shows comparatively low completeness for such messages even though homogeneity is relatively high. For example, AutoReEngine achieves a homogeneity 0.82 for the Twitter dataset but shows completeness of 0.11 because it splits the responses of `SearchTweets` into 190 different clusters on average instead of the actual 3 clusters only. Similarly, it generates many more clusters for GoogleBook and LDAP datasets and thus achieves very low V-measure in message clustering. However, it shows comparatively high accuracy for SOAP datasets. This is because, none of the response messages of the SOAP dataset contain *repetition* of message fields where it shows 1.00 in homogeneity and 0.77 in completeness.

As the results show, our approach accurately clusters the messages with varying length payloads and *repetitive patterns* of payloads by utilizing the position information of keywords. The V-measure in message clustering of our approach is 1.00 for all datasets. This implies that our approach can successfully address the keyword repetition issue and the ambiguity between the true and false keywords.

Format Extraction

Table 5.8 reports the overall results (message-volume weighted) of the response format extraction for the four datasets (More detailed results of format extraction containing the accuracy of the inferred format for the response messages of each request-type based cluster are presented in Appendix A.3). Note that we also report the average number of formats extracted for each cluster for all datasets and ideally it should be 1. The accuracy of the response message format extraction is highly depending on three factors: (i) response message clustering, (ii) format generalization, and (iii) keyword extraction.

As all approaches infer response formats after clustering the response messages, the result of clustering has a huge impact on the inferred message format. More specifically, the message-mixing, *i.e.*, mixing different types of response messages in

Table 5.8: Response Format Extraction Result

Data Set	ProDecoder				AutoReEngine				Our Approach			
	P	R	F	N	P	R	F	N	P	R	F	N
LDAP	0.81	0.91	0.86	1.00	0.83	0.89	0.86	1.05	1.00	1.00	1.00	1.00
SOAP	0.82	0.90	0.86	1.00	0.73	0.84	0.78	1.50	1.00	1.00	1.00	1.00
Twitter	0.77	0.89	0.83	1.00	0.82	0.95	0.88	24.00	1.00	0.99	0.99	1.00
GoogleBooks	0.99	0.81	0.89	1.00	0.96	0.93	0.94	19.00	1.00	1.00	1.00	1.00

*Note: P is precision, R is recall, F is F-measure and N is the average number of formats per ground-truth format.

a cluster, will affect the precision of the inferred format, whereas over-classification, *i.e.*, generating multiple clusters for a single type of responses, will affect the recall of the inferred format. The inferred formats from AutoReEngine are too specific as it generates many more clusters than the actual clusters (*i.e.*, over-classification). Even though the clustering result of AutoReEngine is low, it shows a comparatively good result in format extraction because the evaluation metrics, *i.e.*, *true positive*, *false positive* and *false negative* are calculated in an "`OR`"ed fashion. For example, we count as *true positive* if a test message has been accepted by *any* of the inferred formats. AutoReEngine infers too many formats for the Twitter and GoogleBooks datasets, *i.e.*, it creates 24 clusters on average for the Twitter dataset and 19 for the GoogleBooks rather than the expected 1 cluster. This indicates that AutoReEngine extracts many more false keywords from the messages containing *repetition* sequences of keyword-payload (*i.e.*, in large payloads) and consequently, generates many more clusters than the actual. On the other hand, ProDecoder classifies the messages inaccurately, *i.e.*, mixing up multiple messages into a single cluster due to the misidentification of keywords (*i.e.*, missing true keywords and extracting false keywords) and consequently, infers imprecise message formats (*i.e.*, low precision and recall for the inferred format). In contrast, our approach shows almost 100% precision and recall in the format extraction as it clusters the response messages accurately.

Furthermore, the generalization ability of our approach plays an important role in inferring the response formats and consequently, improves the accuracy of the inferred formats from the response messages. For example, the LDAP response message **`Resp#7`** (in Table 5.1) contains multiple occurrences of '**`ou`**' in the message. Moreover, the whole structure of *search entry* (*i.e.*, keyword-payload sequence) appears repetitively (2 times) in **`Resp#7`**, whereas *search entry* appears only 1 time in **`Resp#6`**. This suggests that the responses of a *search* request may have one or more *search entries*. Therefore, the generalization ability of such *repetitive* patterns allows the inferred format to accept the valid but yet unseen messages. As ProDecoder

adopts MSA to infer the message format for each cluster, it is unable to capture such generalization in the extracted format. As a consequence, it does not achieve 100% precision and recall in format extraction for any of the datasets. Most importantly, it would not be able to achieve 100% accuracy in format extraction even though it would achieve 100% accuracy in clustering due to its inability in handling the *repetition* of keyword-payload sequences. Similarly, AutoReEngine infers the message format as a series of keywords and is unable to capture the generalization of the repetitive patterns in messages. As a result, it achieves low accuracy in format extraction even though it shows high homogeneity in clustering for the GoogleBooks dataset. In contrast to ProDecoder and AutoReEngine, our approach can capture the repetitive patterns and has the generalization capability (*cf.* the regular expression given at the end of Section 5.2). Thus, it achieves 100% precision and recall in format extraction for the LDAP, SOAP and GoogleBook datasets.

In addition, the keyword extraction has an impact on the accuracy of format extraction especially when the extracted keywords are not only used for message clustering but also used for the process of inferring the message formats. Our approach infers format from the *tokenized* messages, where extracted keywords have been used to tokenize the response messages. AutoReEngine infers the format as the series or sequence of keywords from the extracted keywords. Thus, the accuracy of the inferred format for those approaches depends on the extracted keywords from the response messages. Our approach achieves 100% F-measure for all the datasets except for the Twitter dataset, where it achieves 99% recall. This suggests that very few messages are not accepted by the inferred format (*i.e.*, *false negative*). Due to the cross-validation experiment, our approach infers formats of the messages from the messages in the training dataset and then evaluate the inferred formats against the messages in the testing dataset. For the response messages of *statusesshow* sub-cluster, our approach extracts a keyword "`favorite_count:1`", where the number (*i.e.*, '1') is extracted as part of the keyword as all response messages in the training dataset contain such number. But, the response messages (some) in the testing dataset do not contain such number with the keyword "`favorite_count:`" and hence, the response messages in testing dataset are rejected by the inferred format (*i.e.*, *false negative*). Similarly, such keywords in the response messages of the Twitter dataset has also affected the accuracy of the inferred format by AutoReEngine resulting more clusters and message formats.

5.3.2 Efficiency

Again, our main objective is to achieve high accuracy in message format extraction, we also quantify the impact on the efficiency of our approach. We run our experiments and the compared techniques on the same machine as described in Chapter 4 and the time required for clustering the response messages and extracting formats of the response messages are recorded. Table 5.9 shows the average[2] time (seconds) for clustering the response messages and inferring the formats of the response messages, while the detailed results are presented in A.4. As Table 5.9 shows, our approach clusters and infers formats of the response messages faster than the compared techniques for Twitter and GoogleBooks datasets. This is because ProDecoder requires more time in extracting the distributions of the keywords over messages for longer length response messages, while the keyword extraction (*i.e.*, breaking messages into *n*-grams and identifying *n+1* length keyword series) in AutoReEngine requires more time. On the other hand, our approach takes slightly more time in clustering the responses for LDAP and SOAP dataset. This is because our approach extracts keywords with small variations in their positions, which are merged in the keyword merging step based on the standard deviations of their positions. As the payloads (in terms of their length) do not vary much in the LDAP and SOAP datasets compared to the Twitter and GoogleBooks dataset, our approach requires relatively more time in clustering the responses. However, our approach achieves 100% accuracy in clustering and inferring formats of the response messages by sacrificing some efficiency. Again, the accuracy of the inferred format is much important than the efficiency in this thesis, and in particular, the format inference is done in the offline phase before the inferred formats are used in response generation (online) phase.

Table 5.9: Average Time (in seconds) of Inferring Response Format

Data Set	**ProDecoder**		**AutoReEngine**		**Our Approach**	
	T_C	T_F	T_C	T_F	T_C	T_F
LDAP	20.12	2.11	1.47	100.33	40.32	5.21
SOAP	39.12	3.99	19.30	28.67	65.04	39.07
Twitter	1403.55	1839.24	173.31	1148.57	6.74	271.21
GoogleBooks	1907.15	42.28	808.73	1328.02	30.09	2.20

*Note: T_C is the required time (s) for clustering and T_F is the required time (s) for inferring formats.

[2]Each request-based cluster contains different types of responses and we run the 10-fold cross-validation technique for evaluation.

5.4 Summary

In this chapter, we have presented our approach for extracting response formats from the response messages with *repetitive* sequences of keyword-payload. It neither assume any prior knowledge about message structures, nor requires access to the executable code of the applications implementing the services concerned. It takes an interaction trace grouped into request-based clusters as an input and infers the formats of the response messages after clustering the response messages based on their positional keywords. The response messages of each request-based cluster are tokenized based on their positions and then clustered based on these positional keywords. Particularly, it addresses the issues of identifying keywords (*i.e.*, missing true keywords and extracting false keywords) and refines the clusters with mixed message types to get more accurate message clustering for messages with overwhelming payloads. Finally, it infers the formats from the tokenized response messages for each response cluster after extracting cluster-level keywords. We have compared our approach with two existing state-of-the-art approaches for datasets collected from four real services. The experimental results have shown that our approach outperforms existing approaches in both clustering messages and extracting formats from the response messages.

Chapter 6

Inference of Service Behavior

In Chapter 4, we have presented our approach to identifying request message type and inferring formats of each type of messages. In Chapter 5, we have presented our approach to inferring response message formats for each type of response messages in each request-type based cluster. As such, we are now able to identify different types of requests that are involved in a service and to identify different types of responses that are generated for different types of requests and even for the same type of request. As described in Section 2.3.3, the responses of a stateful service depend on the sequence of the preceding interactions and the incoming request messages. For example, a **`search`** request, following an **`add`** request in LDAP, generates a different response than the response generated for a **`search`** request following a **`delete`** request. In addition to the dependency between messages, there also between message fields of a response message and the corresponding request message or even the sequence of the preceding messages.

In this chapter, we propose a new approach to discovering the service behavior model as a state machine through mining the interaction sequences to track the service state and generate the type of response based on the current state of the service for stateful services. We also infer the dependency between important message fields and utilize it in synthesizing response message fields with the aim to generate more accurate responses with appropriate values for the message fields.

This chapter is organized as follows: A simplified form of an example interaction trace is presented in Section 6.1.1 to illustrate our approach in the following sections. The detail of our approach to inferring the service behavior model (message dependency and data dependency models) and utilizing them in synthesizing responses is presented in Section 6.2. The technique of inferring probabilistic service behavior

for the case where the service is not in a clean start state when trace collection starts. In Section 6.4, we present the evaluation of our approach on a number of interaction traces collected from both stateful and stateless services to demonstrate the suitability of our approach in both services. We discuss the limitations of our approach in Section 6.5 before summarizing this chapter in Section 6.6.

6.1 Preliminaries

In this section, we define the key concepts and terms that are used in describing our approach to stateful service virtualization. Section 6.1.1 presents the key concepts and an example interaction trace and Section 6.1.2 presents the result of clustering and inferring message formats from the interaction trace using the techniques described in Chapter 4 and Chapter 5.

6.1.1 Interaction Trace

Table 6.1 shows a simplified form of an example interaction trace for illustrating the approach (without loosing generality) from the CA IM [1] service implementing the widely used LDAP protocol [20]. Instead of using the original form of LDAP messages as presented in Section 2.3.2, we use a simplified form in this Chapter because it is short in length and easier to follow, and yet includes the important message fields. The interactions in Table 6.1 are uniquely identifiable with the key field `cn:`(common name).

The interaction trace in Table 6.1 contains five types of operations: **`Bind`**, **`Add`**, **`Search`**, **`Delete`** and **`Unbind`**, which have been presented in the trace as **`op:B`**, **`op:A`**, **`op:S`**, **`op:D`**, and **`op:U`** respectively. The response operations of **`bind`**, **`add`**, **`search`** and **`delete`** requests are specified with the fields **`op:BindRsp`**, **`op:AddRsp`**, **`op:SearchRsp`** and **`op:DeleteRsp`** respectively, while **`unbind`** request returns *null* response. The result codes for those responses are specified as **`result:Ok`**, **`result:Not found`**, and **`result:AlreadyExists`**. Moreover, we use only the **`cn`** entry to identify an entry or *record* in the data storage uniquely (henceforth being referred as *key field* and its value as *key payload* in this thesis), while in practice, the **`dn`** entry is used to identify an entry or record in the LDAP that contains a **`cn`**, at least one **`ou`** followed by one or more **`dc`** (*cf.* Table 2.1).

The requests and responses in Table 6.1 are executed on four different key payloads *i.e.*, **`cn:Judith`**, **`cn:Gavin`**, **`cn:Linden`**, and **`cn:Katy`**. As Table 6.1 shows, the CA IM service supports different types of requests, *e.g.*, **`Add`**, **`Search`**, and generates

Table 6.1: Simplified Interaction Trace

Index	Request	Response
1	{id:1,op:B,pwd:1234}	{id:1,op:BindRsp,result:Success}
2	{id:2,op:D,cn:Judith}	{id:2,op:DeleteRsp,result:Not Found}
8	{id:8,op:D,cn:Gavin}	{id:8,op:DeleteRsp,result:Not Found}
15	{id:15,op:A,cn:Judith}	{id:15,op:AddRsp,result:Ok}
23	{id:23,op:S,cn:Gavin}	{id:23,op:SearchRsp,result:Not Found}
32	{id:32,op:A,cn:Judith}	{id:32,op:AddRsp,result:Already Exists}
55	{id:55,op:D,cn:Judith}	{id:55,op:DeleteRsp,result:Ok}
90	{id:90,op:D,cn:Gavin}	{id:90,op:DeleteRsp,result:Not Found}
112	{id:112,op:A,cn:Gavin}	{id:112,op:AddRsp,result:Ok}
130	{id:130,op:S,cn:Gavin}	{id:130,op:SearchRsp,result:Ok,cn:Gavin, sn:MAJOR,mobile:26952135}
135	{id:135,op:S,cn:Linden}	{id:135,op:SearchRsp,result:Not Found}
144	{id:144,op:S,cn:Judith}	{id:144,op:SearchRsp,result:Not Found}
210	{id:210,op:A,cn:Gavin}	{id:210,op:AddRsp,result:Already Exists}
213	{id:213,op:A,cn:Linden}	{id:213,op:AddRsp,result:Ok}
235	{id:235,op:A,cn:Judith}	{id:235,op:AddRsp,result:Ok}
242	{id:242,op:A,cn:Katy}	{id:242,op:AddRsp,result:Ok}
251	{id:251,op:S,cn:Judith}	{id:251,op:SearchRsp,result:Ok,cn:Judith, sn:GIDDINGS,mobile:78675623}
283	{id:283,op:A,cn:Gavin}	{id:283,op:AddRsp,result:Already Exists}
300	{id:300,op:U}	

different responses for different types of requests, *e.g.*, **`AddRsp`** and **`SearchRsp`**. The response codes (*i.e.*, results) are also different for different types of responses, *e.g.*, **`Not Found, Ok, Already Exists`**.

6.1.2 Result of Message Analysis

As the basis for further discussions in this chapter, this section presents the result of clustering the interactions and inferring the formats for the messages in Table 6.1 using the techniques described in Chapter 4 and Chapter 5.

Request Based Interaction Clustering:

Tables 6.2, 6.3, 6.4, 6.5, and 6.6 show the interactions after clustering the interactions of Table 6.12, corresponding to the request type **`B`**, **`A`**, **`D`**, **`S`**, and **`U`** respectively

using the technique described in Chapter 4. Table 6.7 shows the inferred formats of the request messages for each request-type based cluster.

Table 6.2: Cluster 1 (`Bind`)

Index	Request	Response
1	{id:1,op:B,pwd:1234}	{id:1,op:BindRsp,result:Success}

Table 6.3: Cluster 2 (`Add`)

Index	Request	Response
15	{id:15,op:A,cn:Judith}	{id:15,op:AddRsp,result:Ok}
32	{id:32,op:A,cn:Judith}	{id:32,op:AddRsp,result:Already Exists}
235	{id:235,op:A,cn:Judith}	{id:235,op:AddRsp,result:Ok}
112	{id:112,op:A,cn:Gavin}	{id:112,op:AddRsp,result:Ok}
210	{id:210,op:A,cn:Gavin}	{id:210,op:AddRsp,result:Already Exists}
283	{id:283,op:A,cn:Gavin}	{id:283,op:AddRsp,result:Already Exists}
213	{id:213,op:A,cn:Linden}	{id:213,op:AddRsp,result:Ok}
242	{id:242,op:A,cn:Katy}	{id:242,op:AddRsp,result:Ok}

Table 6.4: Cluster 3 (`Delete`)

Index	Request	Response
2	{id:2,op:D,cn:Judith}	{id:2,op:DeleteRsp,result:Not Found}
55	{id:55,op:D,cn:Judith}	{id:55,op:DeleteRsp,result:Ok}
8	{id:8,op:D,cn:Gavin}	{id:8,op:DeleteRsp,result:Not Found}
90	{id:90,op:D,cn:Gavin}	{id:90,op:DeleteRsp,result:Not Found}

Table 6.5: Cluster 4 (**Search**)

Index	Request	Response
144	{id:144,op:S,cn:Judith}	{id:144,op:SearchRsp,result:Not Found}
251	{id:251,op:S,cn:Judith}	{id:251,op:SearchRsp,result:Ok,cn:Judith, sn:GIDDINGS,mobile:78675623}
23	{id:23,op:S,cn:Gavin}	{id:23,op:SearchRsp,result:Not Found}
130	{id:130,op:S,cn:Gavin}	{id:130,op:SearchRsp,result:Ok,cn:Gavin, sn:MAJOR,mobile:26952135}
135	{id:135,op:S,cn:Linden}	{id:135,op:SearchRsp,result:Not Found}

Table 6.6: Cluster 5 (**Unbind**)

Index	Request	Response
300	{id:300,op:U}	

Table 6.7: Inferred Formats from the Request Messages

Cluster	Request Formats
Cluster 1 (**Bind** cluster)	{id:1,op:B,pwd:1234}
Cluster 2 (**Add** cluster)	{id:(.*),op:A,cn:(.*)}
Cluster 3 (**Delete** cluster)	{id:(.*),op:D,cn:(.*)}
Cluster 4 (**Search** cluster)	{id:(.*),op:S,cn:(.*)}
Cluster 5 (**Unbind** cluster)	{id:300,op:U}

Response Based Interaction Clustering:

Table 6.8 and 6.9 show the interactions after clustering the interactions of Table 6.5 (**Search**) based on responses, corresponding to **result:Not Found** and **result:Ok** respectively. After clustering the interactions based on the response messages, we infer the format of the response messages for each response based cluster. Table 6.10 shows the inferred response formats from the response cluster 1 (Table 6.8) and cluster 2 (Table 6.9). Then, we store all the different response formats with their corresponding interaction type in a map of key-value pairs (henceforth referred to ResponseMap, where the *interaction type*[1] is the key and the inferred format is the

[1]Interaction type is defined as the combination of the request type and corresponding response type. In case, either the request or the response messages do not contain the type information, the combination of the cluster ID for the request and corresponding responses is considered as the interaction type.

value) to be used in generating responses at runtime. Table 6.11 shows the entries in a ResponseMap for the example interaction trace in Table 6.1.

Table 6.8: `Search` Response Cluster 1 (`Not Found` cluster)

Index	Request	Response
144	{id:144,op:S,cn:Judith}	{id:144,op:SearchRsp,result:Not Found}
23	{id:23,op:S,cn:Gavin}	{id:23,op:SearchRsp,result:Not Found}
135	{id:135,op:S,cn:Linden}	{id:135,op:SearchRsp,result:Not Found}

Table 6.9: `Search` Response Cluster 2 (`Ok` cluster)

Index	Request	Response
251	{id:251,op:S,cn:Judith}	{id:251,op:SearchRsp,result:Ok,cn:Judith, sn:GIDDINGS,mobile:78675623}
130	{id:130,op:S,cn:Gavin}	{id:130,op:SearchRsp,result:Ok,cn:Gavin, sn:MAJOR,mobile:26952135}

Table 6.10: Inferred Formats from the Response Messages

Cluster	Format (Response)
Cluster 1 (`Not Found`)	{id:(.*),op:SearchRsp,result:Not Found}
Cluster 2 (`Ok`)	{id:(.*),op:SearchRsp,result:Ok,cn:(.*), sn:(.*), mobile:(.*)}

Table 6.11: Entries of ResponseMap Containing Response Formats

Key (Interaction Type)	Value (Response Format)
S_SearchRsp(Not Found)	{id:(.*),op:SearchRsp,result:Not Found}
S_SearchRsp(Ok)	{id:(.*),op:SearchRsp,result:Ok,cn:(.*), sn:(.*),mobile:(.*)}
A_AddRsp(Ok)	{id:(.*),op:AddRsp,result:Ok}
A_AddRsp(Already Exists)	{id:(.*),op:AddRsp,result:Already Exists}
D_DeleteRsp(Ok)	{id:(.*),op:DeleteRsp,result:Ok}
D_DeleteRsp(Not Found)	{id:(.*),op:DeleteRsp,result:Not Found}

6.2 Approach

In this section, we present the details of our approach which takes an interaction trace as input and infers the service behavior which can be used to synthesize responses for the incoming requests. It facilitates the creation of "virtual" services for providing test-bed environments by examining *how* actual services respond to the requests. As shown in Figure 6.1, our approach involves three stages: the trace analysis and service behavior inference phase, which are performed offline, and a response generation phase, which is performed online or at runtime.

Trace Analysis: As described in Section 6.1.2, the formats of the request and response messages are extracted for each request based cluster and response based cluster respectively.

Service Behavior Inference: In this step, we infer the behavior of the service from the interaction trace that captures the dependency between messages (henceforth being referred as *message dependency* and *control dependency*, interchangeably) and between message fields, which can be used in synthesizing responses. At first, the interaction trace is partitioned based on the data values of *key fields* (*i.e.*, record in the data storage) that are associated with the request messages. The intuition behind the partitioned to have the relevant interactions sequentially, *i.e.*, creating a separate partition for the interactions operate on the same record. Then, we define the *interaction type* by combining the request type and the corresponding response type for each interaction. The interaction type trace is generated by putting the interaction types together in order as the interactions appear in the record-based partition and the model trace is created by combining the interaction type traces for all record-based partitions. The message dependency model is then inferred from the model trace through generalizing the observed behavior of the individual record-based partition to deduce the process of changing the service state and therefore, to

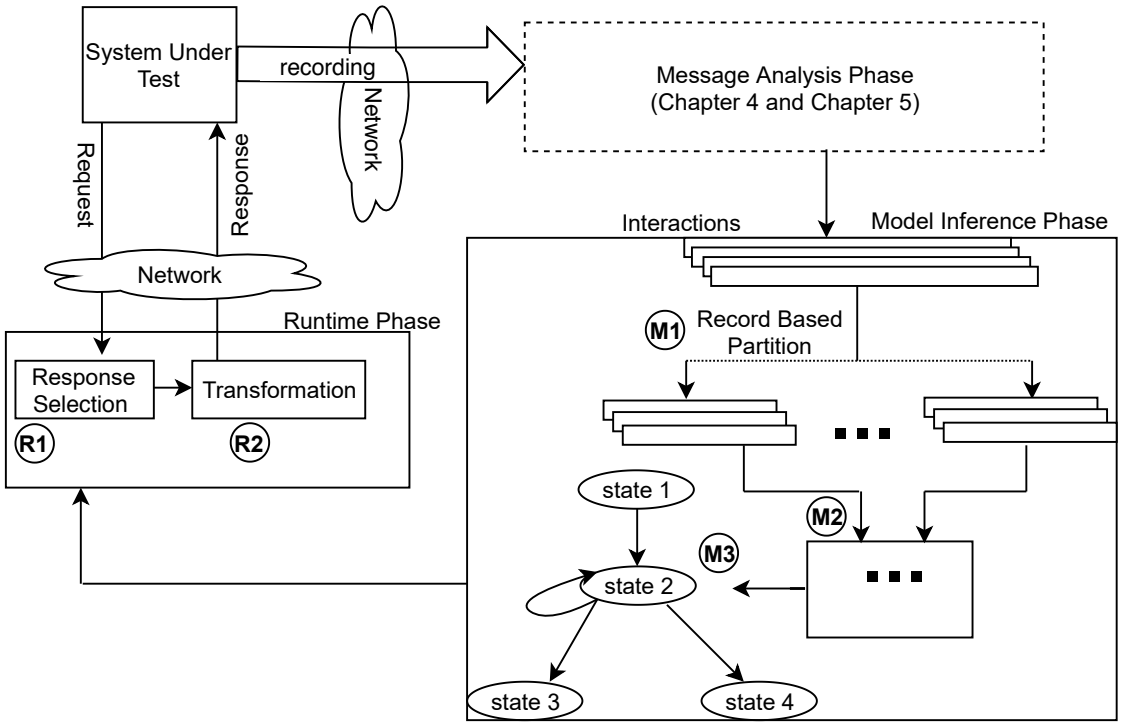


Figure 6.1: Overview of Response Generation Approach

apprehend the reasoning for generating different responses at different times. Then, A set of substitution rules (*i.e.*, data dependency) are inferred for each response-based cluster from the formats of the response and corresponding request messages, and the payloads associated in the messages to determine the payloads for the synthesized response messages.

Response Generation: At runtime, the inferred message dependency model is used to track the state of the service and to identify the type of responses to be sent for the incoming request messages. An individual instance of the inferred model is used for each of the different records or key payloads associated with the request messages to keep track of the service state at runtime. Finally, the inferred payload relationships (*i.e.*, substitution rules) are used to insert the appropriate data values in synthesizing responses and the synthesized responses are sent back for the incoming request messages.

6.2.1 Record Based Partition (Step M1)

In this step, the interaction trace is partitioned based on the key payloads (*i.e.*, record) present in the request messages for having the relevant interactions in the consecutive order. The intuition behind the partition is that *the state of the service is record specific and usually, the operation on one record does not affect the state of*

another record (Section 2.3.3). As Table 6.1 shows, the same **Delete** request executed on two different records (*i.e.*, **Judith** and **Gavin**) generates two different responses (Index:55 and Index:90) due to the distinct service states for the records. It implies that at any time the current state for the key payload "**cn:Judith**" is different from the state for the key payload "**cn:Gavin**" and the operations on "**cn:Judith**" do not change or update the state for the key payload "**cn:Gavin**". As the interactions on different records are interleaved in the interaction trace and our goal is to capture the contextual information to find the previous request or sequence of requests for which a given type of responses are generated, we partition the interaction trace based on the records (*i.e.*, key payload).

Table 6.12: Partitioned Interaction Trace Based on the Key Payloads

Index	Request	Response
2	{id:2,op:D,cn:Judith}	{id:2,op:DeleteRsp,result:Not Found}
15	{id:15,op:A,cn:Judith}	{id:15,op:AddRsp,result:Ok}
32	{id:32,op:A,cn:Judith}	{id:32,op:AddRsp,result:Already Exists}
55	{id:55,op:D,cn:Judith}	{id:55,op:DeleteRsp,result:Ok}
144	{id:144,op:S,cn:Judith}	{id:144,op:SearchRsp,result:Not Found}
235	{id:235,op:A,cn:Judith}	{id:235,op:AddRsp,result:Ok}
251	{id:251,op:S,cn:Judith}	{id:251,op:SearchRsp,result:Ok,cn:Judith, sn:GIDDINGS,mobile:78675623}
8	{id:8,op:D,cn:Gavin}	{id:8,op:DeleteRsp,result:Not Found}
23	{id:23,op:S,cn:Gavin}	{id:23,op:SearchRsp,result:Not Found}
90	{id:90,op:D,cn:Gavin}	{id:90,op:DeleteRsp,result:Not Found}
112	{id:112,op:A,cn:Gavin}	{id:112,op:AddRsp,result:Ok}
130	{id:130,op:S,cn:Gavin}	{id:130,op:SearchRsp,result:Ok,cn:Gavin, sn:MAJOR,mobile:26952135}
210	{id:210,op:A,cn:Gavin}	{id:210,op:AddRsp,result:Already Exists}
283	{id:283,op:A,cn:Gavin}	{id:283,op:AddRsp,result:Already Exists}
135	{id:135,op:S,cn:Linden}	{id:135,op:SearchRsp,result:Not Found}
213	{id:213,op:A,cn:Linden}	{id:213,op:AddRsp,result:Ok}
242	{id:242,op:A,cn:Katy}	{id:242,op:AddRsp,result:Ok}

Table 6.13: Partitioned Interaction Trace Without Key Payload

Index	Request	Response
1	{id:1,op:B,pwd:1234}	{id:1,op:BindRsp,result:Success}
300	{id:300,op:U}	

In this step, we take an interaction trace containing requests and their corresponding responses and a set of user-defined regular expression for identifying the key fields as input. The regular expression is used to parse the request messages and extract the key payloads associated with the interaction (*i.e.*, request message). For example, our approach extracts **Judith** as the key payload from the second interaction (Index:2) of Table 6.1 for a given regular expression **cn=(.*?)**. Similarly, we extract the key payloads from all interactions of an interaction trace. Finally, the interaction trace is partitioned based on the extracted key payloads. In some cases, interactions may not contain such payloads in the request messages (*e.g.*, authentication requests in LDAP). For example, **Bind** and **Unbind** requests in LDAP do not have such key payload, *i.e.*, those messages do not contain the **cn** message field. We create a separate partition containing such interactions of having no key payloads.

Table 6.12 shows the rearranged interaction trace after combining the interactions from the record based partitions, where the interactions of Indices: 2, 15, 32, 55, 144, 235 and 251 are from partition 1 (**Judith**), interactions of Indices: 8, 23, 90, 112, 130, 210 and 283 are from partition 2 (**Gavin**), interactions of Indices: 135, 213 are from partition 3 (**Linden**) and interaction of Index: 242 is from partition 4 (**Katy**). On the other hand, Table 6.13 shows the interactions without key payloads in them (*i.e.*, partition 5).

6.2.2 Model Trace Generation (Step M2)

The purpose of this step is to create a model trace from the interaction trace, which will be used to infer the message dependency model in the model inference step (Step M3). At first, each interaction in the rearranged interaction trace (Table 6.12 and Table 6.13) is marked with the interaction type. For example, the interaction type for the fifth interaction in Table 6.12 (Index: 144) is "**S_SearchRsp(Not Found)**"[2] as the request message belongs to the "**Search**" cluster (Table 6.5) and the corresponding response message is a member of the "**Not Found**" cluster (Table 6.8). A separator

[2]The result "**Not Found**" is used as the response type for the illustration purpose. In practice, the cluster ID of the response message is used as the response type in formulating interaction type. The interaction type for the fifth interaction in Table 6.12 (**Index: 144**) is "**S_Response(Cluster 1)**"

"_" is used in between the request type and the corresponding response type in formulating the interaction type.

Table 6.14: Model Trace (Key Payload)

Trace No.	Traces
1	D_DeleteRsp(Not Found), A_AddRsp(Ok), A_AddRsp(Already Exists), D_DeleteRsp(Ok), S_SearchRsp(Not Found), A_AddRsp(Ok), S_SearchRsp(Ok)
2	D_DeleteRsp(Not Found), S_SearchRsp(Not Found), D_DeleteRsp(Not Found), A_AddRsp(Ok), S_SearchRsp(Ok), A_AddRsp(Already Exists), A_AddRsp(Already Exists)
3	S_SearchRsp(Not Found), A_AddRsp(Ok)
4	A_AddRsp(Ok)

Table 6.15: Model Trace (Non-key Payload)

Trace No.	Traces
1	B_BindRsp(Success), U_""

Once the interactions in the rearranged interaction trace are labeled with the interaction type, an interaction type trace for each record based partition is created by putting the interaction type in a sequential order with a separator (,) as they appeared in the record based partition. For example, the trace in the first row of Table 6.14 is created for the interactions of partition 1 (*i.e.*, interactions of the key payload "**Judith**") and similarly, second, third and fourth traces are created from the interactions of partitions 2, 3 and 4 respectively. Finally, the model trace is generated as a collection of interaction type traces, where we put all interaction type traces for different record-based partitions together. In the same way, another model trace is generated from the *non-key payload* interactions, for example, **Bind** and **Unbind** interactions in LDAP. In contrast to the interaction trace, the model trace only contains the type information of request and response messages, which allows finding the responsible sequence of requests for generating different responses, that is, to extract the dependency between messages and, hence, identify the accurate response type to send for an incoming request.

Table 6.14 shows the generated model trace from the rearranged interaction trace of Table 6.12 (*i.e.*, interactions with key payloads). Table 6.15 shows the generated model trace for the interactions without *key* payloads, *i.e.*, from the interactions of Table 6.13.

6.2.3 Model Inference (Step M3)

In this step, our goal is to infer the message dependency from the generated model trace in *Step M2* and infer message field dependency from the inferred formats in trace analysis phase. This step has two sub-steps: i) Message Dependency, and ii) Data Dependency (*i.e.*, message field dependency).

Message Dependency

The objective of this step is to infer the FSM (Finite State Machine) from the model trace to express the dependency between messages and consequently, to track of the current state of the services at runtime. We use the kTail (k=0) [77] algorithm to infer the control dependency from the model trace generated in the previous step. kTail is the most basic algorithm for inferring a system behavior model from a sequence of inter-component method calls, and several techniques have been proposed to infer a more precise model from the traces based on the kTail algorithm. The latest approaches use stricter generalization rules in merging states and most of them use temporal in-variants to refine the inferred model, intending to prevent imprecise merging and consequently, infer a more precise system model [59, 79, 80, 82]. Those approaches are useful to infer a model from the system logs to validate the system by checking the sequence or order of method calls. The inferred models summarize and generalize the inter-component method calls to support debugging and verifying the system [79]. The system logs contain the execution sequence of methods of each component of a system and the inferred models from the logs are used to find the errors in the execution sequence of methods, stricter generalization rules are applied for inferring such models. For example, *WriteData()* method of a system opens a connection to the database (openConn), writes data (write) and then close the connection (closeConn). Those methods (*i.e.*, openConn, write, closeConn) are to be executed consecutively for the method *WriteData* to function accurately. Therefore, the model inference techniques use the stricter generalization rules to include those methods in a sequential order in the inferred model.

However, to communicate in a service environment, a client consumes services by sending a request message and getting the response back from the service provider. The client does not need to know the internal details of any component or the execution sequence of methods of any component of a service provider, it only consumes the services and the internal details are considered as black-box to the client. For example, in a banking service, a withdraw request may have a series of method calls, *e.g.*, *checkAccount* to check the account number is valid, *checkBalance* to check

the account has sufficient balance to withdraw. To consume the *withdraw* service, a client does not need to know the internal details, *i.e.*, the sequence of internal method calls and the service provider sends the response to the client even if the client sends the request with an invalid account number or with an invalid withdrawal amount. Moreover, in a service, a client is allowed to send any request at any time (*i.e.*, does not need to follow the order/sequence) and always get the response back from the service provider. The service provider sends an error response if an incorrect request is issued by a client. For example, in LDAP, a client must sends a `Bind` request for authentication before sending `Add`, `Search` or any other request to the LDAP server. But, if a client sends `Add`, `Search` or any other request even before authenticating, it will receive an *authentication error* response from the server. As the client does not need to follow any sequence/order for consuming services and the internal structures of any component of a service is unknown to the client, we use the less restrictive (*i.e.*, kTail with k=0) algorithm to infer the message dependency model (*i.e.*, control dependency) and the inferred model is able to capture the dependency relationships among messages.

Figure 6.2 and Figure 6.3 show the inferred message dependencies from the model trace of Table 6.14 and Table 6.15 respectively. As the example dataset contains a very small number of interactions and it does not have all possible sequences of communications, the inferred model in Figure 6.2 from the example model trace does not comprise all possible paths. But, in practice, the inferred models from the model trace for our experimental dataset are most likely the '`flower model`' (*i.e.*, every node in the model is linked with every other node). We present a complete message dependency model inferred by our approach from a large LDAP trace is given in Appendix A.5.

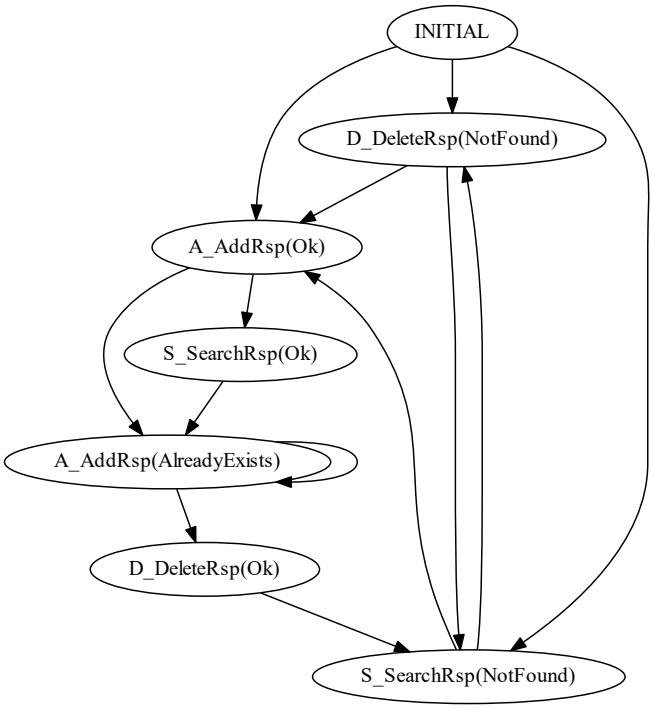


Figure 6.2: Inferred Message Dependency from the Model Trace in Table 6.14

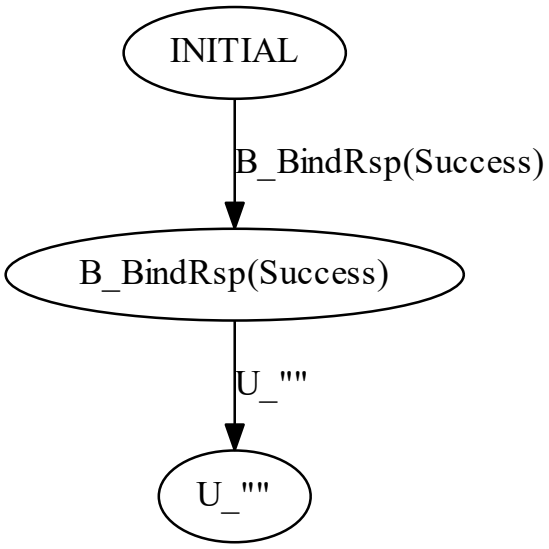


Figure 6.3: Inferred Message Dependency from the Model Trace in Table 6.15

Data Dependency

The objective of this step is to extract the relationships between the message fields as substitution rules from the inferred request format and the corresponding response format. The inferred substitution rules are stored in a map of key-value pairs (henceforth referred to RulesMap, where the interaction type is the key and the substitution rules are the value) for use in synthesizing responses. A set of substitution rules is inferred for each of the response based clusters through comparing the inferred request format and the corresponding request format. At first, a set of candidate substitution rules is inferred by comparing the actual data values (*i.e.*, payloads) of the variable portions of the messages (the request and corresponding response messages). For example, we compare the following request format (*i.e.*, Cluster 3 (**`Search)`** in Table 6.7) with the corresponding format for the responses (*i.e.*, Cluster 2 (**`Ok`**) in Table 6.10)

Request Format : `{id:(.*),op:S,cn:(.*)}`
Response Format : `{id:(.*),op:SearchRsp,result:Ok,cn:(.*), sn:(.*),mobile:(.*)}`

As the request format contains two variable message fields and the response format contains four variable message fields, a maximum of eight (*i.e.*, 2 ×4) different substitution rules can be inferred from these formats. Initially, we infer a candidate substitution rule for each of the variable fields based on the data values for the variable fields of the request messages and for the variable fields of the corresponding responses. For example, our approach extracts **`251`** from the first request message in Table 6.9 (`Index: 251`) for the first variable field of the request format and extracts **`251`** from the corresponding response (`Index: 251` in Table 6.9) for the first variable field of the response format. As the same payload appears in both the request and corresponding response messages, our approach infers a candidate substitution rule for the first variable field of the request and response formats. Then, we verify the inferred candidate substitution rule for the rest of the interactions of the respective cluster and finally, infer the substitution rule only if the candidate substitution rule is valid for all interactions of that cluster. In the same way, our approach infers substitution rules (if possible) for the rest of the variable fields from the above formats. Our approach infers the following substitution rules from the above request and response formats and stores it in the RulesMap for the key "**`S_SearchRsp(Ok)`**"

No.	Substitution Rules
1	{id:(.*?),op:S,cn:} - {id:(.*?),op:SearchRsp,result:Ok,cn:}
2	{,op:S,cn:(.*?)} - {,op:SearchRsp,result:Ok,cn:(.*?)}

Similarly, we infer the substitution rules, *i.e.*, the dependency relationships among message fields for each of the response based clusters and store the inferred rules in the RulesMap. Our approach of inferring substitution rules is more comprehensive compared to the symmetric field identification [2]. This is so because our approach infer candidate substitution rules from an interaction by comparing the payloads in the request message with the payloads in corresponding response message and then verify the candidate rules against all interactions in the respective response-based cluster, whereas the technique presented in [2] identifies symmetric fields by inspecting only the closest matched interaction (*i.e.*, a single interaction). Moreover, our approach infers substitution rules by comparing the message fields of the request and corresponding response formats rather than comparing string (byte) in the symmetric field identification.

Some message fields in the response messages may contain record-specific payloads (*cf.* Section 2.3.3). This suggests that such fields are not directly correlated with the message fields of the corresponding request message. The data dependency model extracts the payloads from the recorded interactions of relevant cluster (based on the incoming request) for such fields in synthesizing responses.

6.2.4 Response Selection (Step R1)

At runtime, the inferred service behavior model (*i.e.*, message dependency and field dependency) is used to formulate the responses for the incoming request messages from the system under test. We utilize the inferred model to find the current state for the record (*i.e.*, key payload) associated with an incoming request and to select the type of response to send back for the incoming request. Algorithm 5 is used to identify the response type by considering the service state for any incoming request at runtime. Here, we use a map to keep the record-specific (payload-specific) history of interactions and to store the record-specific current state of the service. Initially, as shown in Figure 6.2, the current state for all key payloads is set to "`INITIAL`". Now, for any incoming request, we extract the payload associated with the request messages and identify the type of request from the incoming request message at lines *3* and *4* respectively. Then, we search the current state in the map at line *5* for the extracted payload and store it in *c*. If the map does not have any state information for the respective payload, we start with the "`INITIAL`" state of the

Algorithm 5: Response Selection

1: *Input:* request message M, key payload and service state map H, model graph G
2: *Initialization:* current state c=initialState(G), node labels L=getNodeLabel(G)

3: $d \leftarrow getKeyPayload(M)$
4: $r \leftarrow getRequestType(M)$
5: **if** $d \in H.keys$ **then**
6: $\quad c \leftarrow H.get(d)$
7: **end if**
8: **for** $e \in c.outEdges$ **do**
9: $\quad n \leftarrow e.Target$
10: $\quad l \leftarrow L.get(n)$
11: $\quad$ **if** $r \in l$ **then**
12: $\quad\quad insert\ H(d, n)$
13: $\quad\quad Return\ l$
14: $\quad$ **end if**
15: **end for**
16: *Return emptyString*

service. At line *8*, check the node labels of all edges/paths from the current node to identify any node among those connected nodes that contain the request type of incoming message (lines *9* to *11*) and if any such nodes found, the current service state for the associated payload in the map is updated with the new service state and finally, return the response type for the incoming request (lines *12* to *13*). If no such node found (uncommon for a complete model) then return an *empty string* as the response type for the incoming request.

As an example, consider the following `Delete` request as incoming request

```
{id:50,op:D,cn:Dominic}
```

As Algorithm 5 depicts, our approach extracts `Dominic` as the key payload and `D` as the request type. Then it finds the current state for the payload `Dominic` in the map and as the key payload `Dominic` appears for the first time, the current state (c) is set to the "`INITIAL`" state. Now, it checks all of the out edges of the initial state and finds a node among the connected nodes that contains the incoming request (*i.e.*,`D`). Our approach found the connected node `D_DeleteRsp(NOT Found)`, which contains the incoming request type `D`. It then updates the current state for the key payload `Dominic` to "`D_DeleteRsp(NOT Found)`" in the map and finally, returns `D_DeleteRsp(NOT Found)` as the response type for the above incoming request.

6.2.5 Response Transformation (Step R2)

In this step, we synthesize the response once the type of response to send back for the incoming request is identified in the previous step. First, we find the format of the response to send in the ResponseMap (Table 6.11) using the selected response type. For example, if the "`D_DeleteRsp(Not Found)`" response is selected in the response selection step (*i.e.*, *Step R1*) for an incoming `Delete` request, the format of the response to send is "`{id:(.*),op:DeleteRsp,result:Not Found}`". Once we get the format of the response from the ResponseMap, the variable portions of the format need to be filled with the appropriate values (*i.e.*, payloads) and finally, return the synthesized response to the requester (*i.e.*, the system under test). The payloads of the synthesized response can be divided into two categories: i) message specific payloads and ii) key payload (record) specific payloads.

The first category of payloads is used to describe the response message itself. The `message id` field in LDAP messages is such an example and used to uniquely identify the messages and to correlate the request and response messages. It implies that the value of the `message id` field in the response must be the same as in the request message to be considered as a valid response. We use the message field dependency relationships, *i.e.*, inferred substitution rules to insert the appropriate payloads for those message-describing fields in the synthesized responses.

The second category of payloads is related to the *key payloads*. For example, the LDAP search response contains a `mobile number, telephone number, address, email, postal code, surname,` etc., which are information related to an entry (*i.e.*, record). The response in Table 6.12 (`Index: 251`) contains `mobile number` and `surname`, which are related information of key payload `cn:Judith`. As our approach does not consider general *data model* for generating data values, we use the payloads from one of the responses in the interaction trace to approximate the data values for those fields in the generated responses. In this step, we randomly select one of the interaction from the respective response cluster because the closest matched interaction as described in [9] does not improve the accuracy although reduces the efficiency. For example, if the `D_DeleteRsp(Not Found)` response is selected in Step R1 for an incoming `Delete` request then we select one interaction randomly from the `DeleteRsp(Not Found)` cluster. After choosing the interaction, we extract the payloads from the response of the chosen interaction and insert them into the synthesized responses for the record-specific fields and finally, send the synthesized responses to the system under test.

For the example incoming request, `D_DeleteRsp(Not Found)` is selected as the

response type in the previous step. Now, we search for the response format in the ResponseMap (*i.e.* Table 6.11) using the key "`D_DeleteRsp(Not Found)`" and also search for the substitution rules in the RulesMap with the same key to insert the appropriate payloads into the synthesized response. We get the following response format and substitution rules in the respective maps for the response type `D_DeleteRsp(Not Found)`:

Response Format: `{id:(.*),op:DeleteRsp,result:Not Found}`

Substitution Rule: `{id:(.*?),op:D,cn:} - {id:(.*?),op:DeleteRsp,result:Not Found}`

As the substitution rule contains two different portions, we use the first portion of the substitution rule (*i.e.*, `{id:(.*?),op:D,cn:}`) to extract the payload from the incoming request message and then use the second portion of the substitution rule (*i.e.*, `{id:(.*?),op:DeleteRsp,result:Not Found}`) to insert the payload into the response message. Our approach extracts "`50`" from the example incoming request and the following response will be sent back after inserting the extracted payload (*i.e.*, "`50`") into the response format and the generated response is *identical* to the expected response.

`{id:50,op:DeleteRsp,result:Not Found}`

Let us consider another example: an incoming request `Search`, followed by `Delete` and `Add` requests. The incoming request and the expected or actual response are as below:

Incoming Request: `{id:345,op:S,cn:Craig}`
Expected Response: `{id:345,op:SearchRsp,result:Ok,sn:LINK,mobile:543219087}`

The response type `S_SearchRsp(Ok)` is selected from the model according to *Step R1*. We get the following response format and substitution rules (two) respectively for the response type `S_SearchRsp(Ok)`

Response Format: `{id:(.*),op:SearchRsp,result:Ok,cn:(.*),sn:(.*),mobile:(.*)}`

Substitution Rules: `{id:(.*?),op:S,cn:} - {id:(.*?),op:SearchRsp,result:Ok,cn:}`
`{,op:S,cn:(.*?)} - {,op:SearchRsp,result:Ok,cn:(.*?)}`

As the response format contains four variable message fields, the appropriate payloads (values) need to be inserted for those fields in the synthesized response. Our approach utilizes the inferred substitution rules to get the respective payloads from the incoming request and insert them into the synthesized response. Our approach extracts "`345`" and "`Craig`" from the incoming request using the first and second substitution rules respectively and generates the following response after inserting them into the response

```
{id:345,op:SearchRsp,result:Ok,cn:Craig,sn:(.*),mobile:(.*)}
```

As the generated response does not have data values for its two remaining fields (record-specific payloads), our approach finds an interaction from the `S_SearchRsp(Ok)` cluster (*i.e.*, Table 6.9) *randomly* and extracts the payloads for those fields and finally, insert those payloads in the synthesized response. Let us assume that our approach selects the second interaction from Table 6.9. The following response is synthesized after inserting the data values for `sn` and `mobile` fields

```
{id:345,op:SearchRsp,result:Ok,cn:Craig,sn:MAJOR,mobile:26952135}
```

The above synthesized response contains the expected payloads in `id` and `cn` but not in `sn` and `mobile`.

6.3 Message Dependency (Non-clean Start)

The inferred service behavior model in Figure 6.2 is deterministic as the traces in Table 6.1 are collected based on the assumption that *the initial state of service state is empty*, *i.e.*, none of the records were added before collecting the traces (henceforth being referred as *Clean Start)*). In practice, the initial state of the service is often *non-empty* (henceforth being referred as *non-clean Start*) when data collection starts, as we do not have access to the actual service except some interactions between the actual service and a client. The records presented in the captured interaction traces may or may not be added before data collection begins. Such records play an important role to decide the responses to send for the incoming requests, especially for stateful services. For example, the response for the `Delete` request (`Index: 2`) in Table 6.1 would be `Ok` instead of `Not Found` if the record "`Judith`" was added before recording/capturing the interaction trace and in such case, the subsequent responses for the same request with the record "`Judith`" would also be different than the responses in Table 6.1. Table 6.16 shows the captured interaction

trace when the record "**Judith**" was added before intercepting the communication. As Table 6.16 shows, the same **Delete** request on different records, *i.e.*, on "**Judith**" and "**Gavin**" generates different types of responses. It indicates that the same request may generate two or more different types of responses depending on the initial state of the service. To handle such issues, we infer a probabilistic message dependency model from the interaction traces for describing both the transitions and their probabilities in the inferred model. At runtime, the type of the responses to send for the incoming request messages may be selected (as described in Section 6.2.4) in two different ways, *i.e.*, the transition in the control dependency model is selected: i) randomly, and ii) based on the probabilistic scores associated with the edges. We run the response generation experiments using both ways of selecting the response type from the inferred message dependency model.

Figure 6.4 shows the probabilistic message dependency inferred from the interactions in Table 6.16.

Table 6.16: Interaction Trace (Non-clean Start)

Index	Request	Response
1	{id:1,op:B,pwd:1234}	{id:1,op:BindRsp,result:Success}
2	{id:2,op:D,cn:Judith}	{id:2,op:DeleteRsp,result:ok}
8	{id:8,op:D,cn:Gavin}	{id:8,op:DeleteRsp,result:Not Found}
15	{id:15,op:A,cn:Judith}	{id:15,op:AddRsp,result:Ok}
23	{id:23,op:S,cn:Gavin}	{id:23,op:SearchRsp,result:Not Found}
32	{id:32,op:A,cn:Judith}	{id:32,op:AddRsp,result:Already Exists}
55	{id:55,op:D,cn:Judith}	{id:55,op:DeleteRsp,result:Ok}
90	{id:90,op:D,cn:Gavin}	{id:90,op:DeleteRsp,result:Not Found}
112	{id:112,op:A,cn:Gavin}	{id:112,op:AddRsp,result:Ok}
130	{id:130,op:S,cn:Gavin}	{id:130,op:SearchRsp,result:Ok, sn:MAJOR,mobile:26952135}
135	{id:135,op:S,cn:Linden}	{id:135,op:SearchRsp,result:Not Found}
144	{id:144,op:S,cn:Judith}	{id:144,op:SearchRsp,result:Not Found}
210	{id:210,op:A,cn:Gavin}	{id:210,op:AddRsp,result:Already Exists}
213	{id:213,op:A,cn:Linden}	{id:213,op:AddRsp,result:Ok}

Table 6.16 Continued: Interaction Trace (non-clean Start)

Index	Request	Response
235	`{id:235,op:A,cn:Judith}`	`{id:235,op:AddRsp,result:Ok}`
242	`{id:242,op:A,cn:Katy}`	`{id:242,op:AddRsp,result:Ok}`
251	`{id:251,op:S,cn:Judith}`	`{id:251,op:SearchRsp,result:Ok, sn:GIDDINGS,mobile:78675623}`
283	`{id:283,op:A,cn:Gavin}`	`{id:283,op:AddRsp,result:Already Exists}`
300	`{id:300,op:U}`	

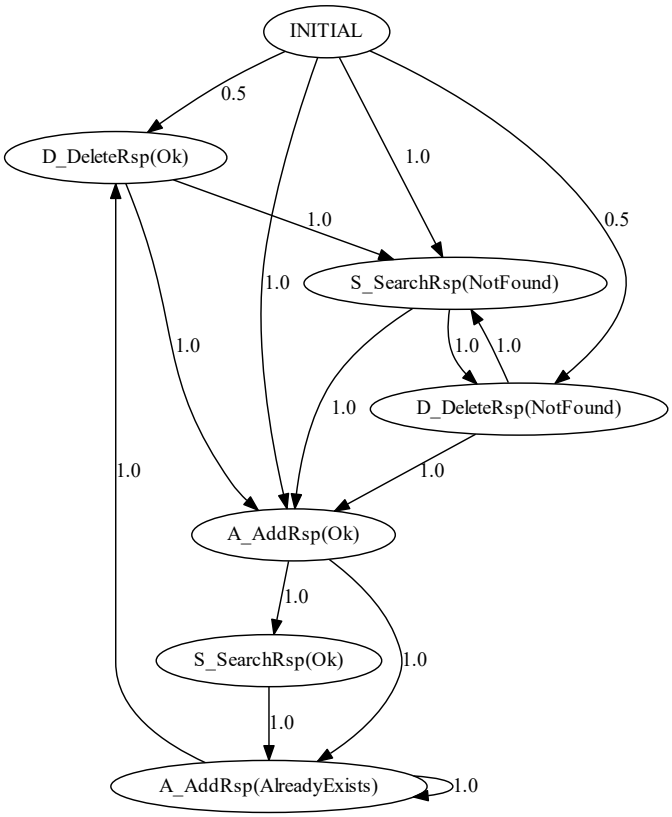


Figure 6.4: Probabilistic Message Dependency Inferred from the Example Interaction Trace

6.4 Evaluation

In this section, we present the experimental results to evaluate the effectiveness of our approach. In Section 6.4.1, we present a summary of the datasets. The

evaluation approach and criteria are presented in Section 6.4.2. The compared techniques are presented in Section 6.4.3, and the results of our experiments are presented in Section 6.4.4.

6.4.1 Datasets

We run the experiments on datasets collected from both stateful (*i.e.*, LDAP and SOAP) and stateless services (*i.e.*, Twitter and GoogleBooks). For stateless services, we use the same datasets as described in Chapter 4. For stateful services, we use different traces based on the initial state of the services to run the experiments. The LDAP datasets contain 20157 and 19930 interactions when the service state is *initial* and *non-initial* respectively, while the SOAP datasets contain 20000 interactions in both cases. We run the experiments on stateless datasets (*i.e.*, Twitter, GoogleBooks) to demonstrate that our approach is applicable for virtualizing stateless services as well.

6.4.2 Evaluation Approach and Criteria

To evaluate the accuracy of our approach, we use the popular *cross-validation* approach [121], with the advantage of validating each interaction exactly once and ensuring that all interactions are used both in training and testing datasets. In our experiments, we use 10-fold cross-validation [109] to the experimental datasets. As we aim to synthesize responses for stateful services and the sequence of interactions affect the current state of the services, we need to keep the relevant interactions (*i.e.*, record-based interactions) together. To do so, we partition the interaction trace based on the key *records* associated with the request messages and generate folds based on the number of record based partitions. As the folds are generated using the interactions from the key record based partitions, not every fold contains the same number of interactions in the training and testing dataset. However, our technique ensures all interactions are used both in the testing and training datasets. We also measure the efficiency in terms of time required to generate responses and the maximum amount of memory required to run our approach and the compared techniques.

To measure the accuracy, we compare the synthesized responses with the recorded or actual responses and the following criteria are used in assessing the synthesized responses:

Table 6.17: Accuracy Criteria for Assessing the Synthesized Responses

(i)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
(ii)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:**RAYMOND**}
(iii)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:**32**,op:SearchRsp,result:Ok,cn:**Dominic**, sn:**RAYMOND**}
(iv)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:**32**,op:SearchRsp,result:**Not found**}
(v)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:32,op:**AddRsp**,result:Ok}
(vi)	Expected	{id:25,op:SearchRsp,result:Ok,cn:Joseph, sn:GRIMES}
	Generated	{id:32,op:**SearAdRsp**,result:Ok}

1. **Identical:** The synthesized response is identical to the expected (or recorded) response (*cf.* Example (i) in Table 6.17)

2. **Data Consistent:** The synthesized response contains the expected response type and has the expected payloads in the critical fields[3] but may differ in some payload information (*cf.* Example (ii) in Table 6.17 where "`id`" and "`cn`" are identical with the expected response but differs in "`sn`")

3. **Protocol Exact:** The synthesized response contains the expected response type, but it differs in the critical and possibly other fields (*cf.* Example (iii) in Table 6.17 where "`op`" (response type) and "`result`" (response code) are identical but differ in critical fields, *i.e.*, "`id`", "`cn`" and other fields, *i.e.*, "`sn`")

4. **Protocol Plausible:** The response type of a synthesized response is identical to that of the expected response but has the wrong response code (*i.e.*, result) and possibly differs in "`id`" (*cf.* Example (iv) in Table 6.17)

5. **Well-Formed:** The synthesized response has the wrong response type (*i.e.*, op) but it corresponds to one of the valid response messages (*cf.* Example (v) in Table 6.17)

6. **Malformed:** Synthesized response does not meet any of the above criteria, *i.e.*, not a valid type of response at all (*cf.* Example (vi) in Table 6.17).

[3]We consider the key fields (*e.g.*, "`cn`" in LDAP) and the message-describing fields (*e.g.*, "`message id`" in LDAP) together as critical fields.

6.4.3 Compared Techniques

We compare our approach with two other existing Opaque Service Virtualization (OSV) approaches of synthesizing responses for stateless services. One of those is the *Whole Cluster* [2] approach, identifying a closest matched interaction from the recorded interactions for the incoming request and send back the responses after transformation. The second compared method is the message prototype based OSV [11]. It uses multiple sequence alignment to generate message prototypes for different types of requests and at runtime, compares the incoming request with the generated prototype instead of the recorded interactions to find the closest matched interaction. Although the technique proposed in [37] aims for stateful service virtualization, we did not compare our approach against it because its limitations as listed in Chapter 3, especially for its inability in generating application layer responses with payloads.

6.4.4 Results

We evaluate our approach in two aspects: *accuracy* and *efficiency*, with the aim to answer the following questions

- **Q1 (Accuracy):** With the inferred service behavior model as described in Section 6.2, is our technique able to generate more accurate responses for both stateless and stateful services? Does the inferred formats as described in Chapter 4 and Chapter 5 and the inferred dependency between message fields as described in Section 6.2.3 improve the accuracy of the synthesized responses?

- **Q2 (Efficiency):** Is our technique able to generate responses in a reasonable (relative to the actual services) time with the use of attainable resources?

Accuracy

This section presents the result in terms of accuracy of our approach and the compared techniques in synthesizing responses. The experiments are carried out on the traces that are collected from two different scenarios: i) Clean Start, the data/records used in collecting traces were not added before data collections starts and ii) Non-clean Start, the records associated with the captured interactions in the trace may or may not be added (*i.e.*, a few records may be added, while others not) before data collections starts.

Clean Start

This section presents the result of the experiments considering *clean start*. Figure 6.5 shows the accuracy of our approach versus opaque service virtualization (OSV) approaches (whole cluster and prototype) in generating responses, while Table 6.18 shows the detailed result of response generation. As Figure 6.5 shows that our approach outperforms the OSV approaches in generating accurate responses for stateful services. For the LDAP dataset, our approach generates about 88% identical responses, while the whole cluster and prototype-based OSV generate about 33% and 30% identical responses respectively. This demonstrates more than 50% improvement over the existing OSV approaches (whole cluster and prototype) in generating *identical responses*. As the Figure 6.5 shows, all three approaches generate data consistent responses for LDAP. It happens because the actual responses for a few requests contain the *record-specific* payloads, *e.g.*, `mobile number`, `address` etc. and none of the techniques consider an accurate "data model" in synthesizing responses and hence, generates responses with the different payloads in such fields. Again, all approaches generate protocol plausible responses for the LDAP dataset. But, our approach generates only about 3% protocol plausible responses, whereas the whole cluster and prototype generate about 43% and 39% respectively. It indicates that our approach reduces the generation of protocol plausible responses compared to other approaches. However, with the use of the service behavior model, our approach is expected to generate either identical or data consistent responses only but it also generates very few protocol plausible responses. The reason for that one of the request operations, *i.e.*, the `ModifyDN` request, updates the value of the key field, causes the inferred model being unable to keep track of the service state for the updated key payload and hence, generates protocol plausible responses for the future requests associated with the updated key payload. The detailed process of generating protocol plausible responses is explained in Appendix A.6 with an example interaction trace. Very importantly, our approach does not generate any malformed responses, while the whole cluster and prototype techniques generate about 7% and 23% malformed responses respectively. Neither the whole cluster approach nor the prototype-based OSV considers the message structure during payload substitution and hence, generates malformed responses. On the other hand, our approach infers the format of the messages and utilizes the inferred format in substituting payloads to synthesizing responses and consequently, does not generate any malformed responses.

Similarly, for the SOAP dataset, our approach generates about 43% identical responses, whereas the whole cluster and prototype generate about 40% and 35% iden-

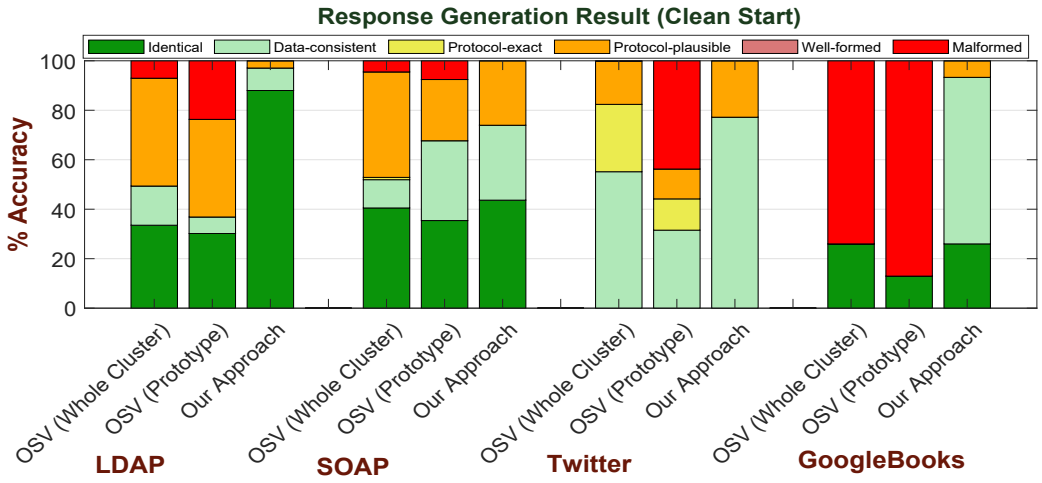


Figure 6.5: Response Generation Result (Clean Start)

tical responses respectively. Again, our approach does not generate any malformed responses, while the compared approaches generate some malformed responses. For SOAP, the improvement in generating identical responses of our approach is less as compared to the LDAP dataset. It shows that our approach generates more data consistent and protocol plausible responses compared to the LDAP dataset and hence, reduces the generation of identical responses. The reason for generating more data consistent responses is that the responses for most of the request messages in SOAP, *i.e.*, `getAccount, withdraw, deposit` contain the value of `balance` field and the value of `balance` field changes over different types of requests. As our approach does not consider an accurate "*data model*", *i.e.*, how payloads change over time in synthesizing responses, the synthesized responses do not contain the exact payloads for the `balance` field. Similarly, the reason for generating more protocol plausible responses is that i) LDAP is a CRUD (Create, Read, Update and Delete) type service, where an entry or record can be deleted with the `delete` request and be re-created with the `add` request, whereas a bank account can only be created/opened once and there is no request in the SOAP trace to re-open the same account after closing it through the `closeAccount` request and ii) the type of the responses (*e.g.*, `DeleteRsp(Success), DeleteRsp(Not Found)`) for the incoming requests in LDAP depend only on the key payload. So, the different types of responses are generated based on whether the record is already added, deleted, etc. in the data store or not, whereas the response type (*e.g.*,`withdrawResponse(Success), withdrawResponse(Fail)`) for a `withdraw` request in SOAP not only depends on the key payload "*account number*", but also depends on the current balance and the requested withdrawal amount. Therefore, the response of a `withdraw` request can be `withdrawResponse(Fail)` either due to the invalid account number or the insufficient

Table 6.18: Response Generation Result (Clean Start)

Dataset	Method	No.	Identical	Data Consistent	Protocol Exact	Protocol Plausible	Well-Formed	Malformed
LDAP	OSV (Whole Cluster)	20157	6760 (33.54%)	3192 (15.84%)	0	8786 (43.58%)	0	1419 (7.04%)
	OSV (Prototype)		6086 (30.19%)	1344 (6.67%)	0	7954 (39.46%)	0	4773 (23.68%)
	Our Approach		17733 (87.98%)	1835 (9.10%)	0	589 (2.92%)	0	0
SOAP	OSV (Whole Cluster)	20000	8101 (40.51%)	2284 (11.42%)	189 (0.94%)	8526 (42.63%)	0	900 (4.50%)
	OSV (Prototype)		7082 (35.41%)	6453 (32.27%)	0	4948 (24.74%)	0	1517 (7.58%)
	Our Approach		8734 (43.67%)	6057 (30.28%)	0	5209 (26.05%)	0	0
Twitter	OSV (Whole Cluster)	1465	0	808 (55.15%)	399 (27.24%)	256 (17.47%)	0	2 (0.14%)
	OSV (Prototype)		0	462 (31.54%)	185 (12.63%)	177 (12.08%)	0	641 (43.75%)
	Our Approach		0	1131 (77.20%)	0	334 (22.80%)	0	0
Google-Books	OSV (Whole Cluster)	1913	497 (25.98%)	0	0	0	0	1416 (74.02%)
	OSV (Prototype)		248 (12.96%)	0	0	0	0	1665 (87.04%)
	Our Approach		497 (25.98%)	1288 (67.33%)	0	128 (6.69%)	0	0

fund in the balance field and hence, our approach is unable to track of the service state due to the *non-key* payload.

In general, our approach achieves a significant accuracy improvement in generating accurate responses compared to the existing techniques for stateful services, *i.e.*, for LDAP and SOAP. our approach considers the current state of the service and utilizes the relationships among message fields in formulating responses for the incoming request messages. Thus, it is less chance to generate inaccurate responses, whereas the whole cluster and prototype approaches do not consider the service state and message structure in synthesizing responses and hence, generate malformed responses.

For stateless services, *i.e.*, for Twitter and GoogleBooks, our approach does not

generate malformed responses, while the compared techniques generate malformed responses due to the same reason as described above (*i.e.*,). As Table 6.18 shows, none of the approaches generate identical responses for the Twitter dataset because the responses contain massive payloads including a timestamp. However, our approach generates about 77% data consistent response, while prototype and whole cluster approach generate about 31% and 55% data consistent responses. As Table 6.18 shows, all three approaches generate protocol plausible responses for the Twitter dataset. It happens because some requests generate different types of responses but the captured interaction trace is not recognized as the reason for generating different responses. For example, the response of a `searchTweets` request can either be empty or a number of Tweets based on the search query but it is not possible to infer the reason for generating different responses by analyzing the interaction trace only. Unlike the Twitter dataset, our approach and the compared techniques generate identical responses for the GoogleBooks dataset. The responses of `search_bookshelf` contain "error" message when the `bookshelf_id` is not found in the GoogleBooks. The "error" responses share exactly the same message structure and hence, all three approaches generate a few identical responses for such requests. Similar to the Twitter dataset, our approach generates protocol plausible responses for the GoogleBooks dataset as a request generates different types of responses at different times but the dependency among messages is not observable in the captured interaction trace.

In general, our approach outperforms the existing service virtualization techniques in generating accurate responses. The message dependency model facilitates to select the correct types of responses for the incoming requests and the substitution rules to insert the appropriate payloads in the synthesized responses. Unlike the whole cluster and prototype-based OSV approaches, our approach stops the generation of malformed responses as it utilizes the inferred message formats in synthesizing responses.

Non-clean Start

Our approach infers a probabilistic message dependency model for non-clean start, *i.e.*, the record may or may not be added before data collection starts. In such case, the types of responses for the incoming requests are selected in two different ways: i) randomly or ii) with the use of probabilistic scores that has been seen in the training dataset.

Figure 6.6 and Table 6.19 show the accuracy of synthesizing responses for non-clean start. As the result shows, our approach (random and probabilistic) achieves higher accuracy in generating identical responses compared to the existing techniques for stateful services (*i.e.*, LDAP and SOAP). Again, the improvement in

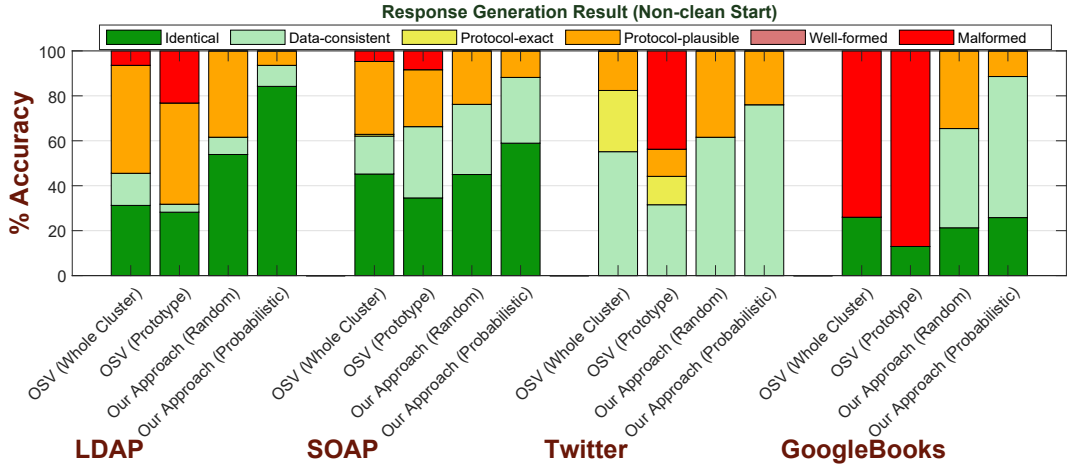


Figure 6.6: Response Generation Result (Non-clean Start)

generating identical responses for the SOAP dataset is not as much as LDAP due to the same reason as described in the previous section (*i.e.*, Clean Start). As Figure 6.6 shows, our approach of selecting the type of response based on the probabilistic score achieves better accuracy for all datasets compared to randomly selecting the type of response from the message dependency model. This is so because our approach (probabilistic) considers the ratios of their appearance (in training dataset) in choosing the response types for the incoming requests.

For stateless services, the change of the service state is not observable in the interaction trace and we use the same dataset, our approach of inferring and utilizing probabilistic message dependency model achieves similar accuracy compared to our approach with deterministic model in synthesizing responses for Twitter and GoogleBooks datasets.

In general, our approach achieves higher accuracy compared to the existing techniques in synthesizing responses for both stateful and stateless services. Again, as the Figure 6.5 and Figure 6.6 show, our approach outperforms the existing approaches for both clean start and non-clean start. First, our approach does not generate any malformed responses as it utilizes the format of the messages in formulating responses. On the other hand, the existing approaches (*i.e.*, the whole cluster and prototype OSV) do not consider the message structure in generating responses and consequently, generate malformed responses. Moreover, the identification and classification of the response messages for the same request allow our approach to generating the protocol plausible responses for the incoming requests. The inferred message dependency model facilitates the generation of identical, data consistent and protocol exact responses.

Table 6.19: Response Generation Result (Non-clean Start)

Dataset	Method	No.	Identical	Data Consistent	Protocol Exact	Protocol Plausible	Well-Formed	Malformed
LDAP	OSV (Whole Cluster)	19930	6226 (31.24%)	2850 (14.30%)	0	9572 (48.03%)	0	1282 (6.43%)
	OSV (Prototype)		5627 (28.23%)	708 (3.55%)	0	8976 (45.04%)	0	4619 (23.18%)
	Our App (Random)		10742 (53.90%)	1532 (7.69%)	0	7656 (38.41%)	0	0
	Our App (Probabilistic)		16784 (84.21%)	1865 (9.36%)	0	1281 (6.43%)	0	0
SOAP	OSV (Whole Cluster)	20000	9042 (45.21%)	3373 (16.87%)	157 (0.78%)	6485 (32.43%)	0	943 (4.71%)
	OSV (Prototype)		6913 (34.57%)	6346 (31.73%)	0	5067 (25.33%)	0	1674 (8.37%)
	Our App (Random)		8998 (44.99%)	6244 (31.22%)	0	4758 (23.79%)	0	0
	Our App (Probabilistic)		11787 (58.94%)	5852 (29.26%)	0	2361 (11.80%)	0	0
Twitter	OSV (Whole Cluster)	1465	0	808 (55.15%)	399 (27.24%)	256 (17.47%)	0	2 (0.14%)
	OSV (Prototype)		0	462 (31.54%)	185 (12.63%)	177 (12.08%)	0	641 (43.75%)
	Our App (Random)		0	902 (61.57%)	0	563 (38.43%)	0	0
	Our App (Probabilistic)		0	1114 (76.04%)	0	351 (23.96%)	0	0
Google-Books	OSV (Whole Cluster)	1913	497 (25.98%)	0	0	0	0	1416 (74.02%)
	OSV (Prototype)		248 (12.96%)	0	0	0	0	1665 (87.04%)

Table 6.19 Continued: Response Generation Result (Non-clean Start)

Dataset	Method	No.	Identical	Data Consistent	Protocol Exact	Protocol Plausible	Well-Formed	Malformed
	Our App (Random)		407 (21.28%)	845 (44.17%)	0	661 (34.55%)	0	0
	Our App (Probabilistic)		494 (25.82%)	1201 (62.78%)	0	218 (11.40%)	0	0

Efficiency

To measure the efficiency of our approach, we instrument the response generation code and record the response generation times for all datasets. We run the experiments on Intel Core(TM) i5-4570 3.20 GHz with 16GB of main memory. For comparison we run our approach and the compared techniques (*i.e.*, the whole cluster and prototype OSV) on the same CPU and also logged the actual time of getting response back from the actual services at the time of collecting the interaction traces.

Response Generation Time

Table 6.20 shows the average time (ms) to generate responses using the whole cluster OSV, prototype OSV and our approach. It also reports the average response generation time for the actual services. As Table 6.20 shows, our approach takes only a fraction of a millisecond to generate responses and significantly faster than the compared approaches and even faster than the actual services. For example, the actual *IM* service takes 55.74ms on average to generate a response for the incoming request, while our approach requires only 0.35ms to generate a response, *i.e.*, 159 times faster than the actual service. On the other hand, the whole cluster approach requires an extremely long time to find the closest matched interaction, as it compares the incoming request against all interactions in the training dataset and the experimental dataset contains a large number of interactions, especially for stateful services. The whole cluster approach is 67 times slower than the actual LDAP and 10653 times slower than our approach in generating responses. As Table 6.20 shows, the whole cluster consumes about 90% of the time or more to find the closest matched interaction through comparing the incoming request message against all interactions in the trace and hence, requires a long time to generate responses. On the other hand, the prototype-based OSV compares the incoming request messages

Table 6.20: Average Response Generation Time (ms)

Dataset	No.	Real Services	OSV (Whole Cluster)			OSV (Prototype)			Our Approach			
			M	S	T	M	S	T	RS	M	S	T
LDAP	19930	55.74	3784.53	1.33	3785.86	2.07	0.37	2.44	0.14	0.01	0.20	0.35
SOAP	20000	16.52	8264.33	1.38	8265.71	2.88	0.81	3.69	0.31	0.01	0.07	0.39
Twitter	1465	278.3	118.69	12.72	131.41	0.39	15.76	16.15	0.05	0.01	0.18	0.24
Google-Books	1913	10.66	42.77	31.98	74.75	2.59	151.88	154.47	0.03	0.01	0.07	0.11

*Note: M is the time for Matching, S is the time for Substitution, RS is the time for the Response Selection and T is the total time.

Table 6.21: Average Analysis Time

Dataset	No.	OSV (Prototype)	Our Approach		
		Analysis (s)	Analysis (s)	Model Inference (ms)	Total (s)
LDAP	19930	8959.49	3254.06	64.69	3254.12
SOAP	20000	95871.22	5590.37	43.46	5590.41
Twitter	1465	18657.06	1086.18	98.31	1086.27
GoogleBooks	1913	21456.32	1821.48	10.36	1821.49

with the generated prototype for each different request instead of comparing it with the actual request messages in the trace. Thus, it generates responses much faster than the actual services and whole cluster approach for all datasets except Google-Books. For example, prototype-based OSV is 22 times faster than the actual service for LDAP and 1551 times faster than the whole cluster approach. However, it is about 7 times slower than our approach to generating responses for LDAP. As the responses in the GoogeBooks dataset contain the "`searched query`" multiple times (depending on the number of books) and prototype approach extracts substitution rules at runtime, the process of synthesizing responses requires a long time and hence, the prototype approach is slow for GoogleBooks. In contrast, our approach infers the substitution rules at the trace analysis phase and utilizes the inferred rules at the time of generating responses and hence, generates response more efficiently compared to the existing approaches.

Table 6.21 reports the average time for the analysis phase. As the whole cluster approach does not have an analysis phase, Table 6.21 shows the analysis time required for our approach and the prototype-based OSV approach. Our approach requires less time for the trace analysis phase compared to the prototype-based approach for all datasets.

Memory Usage

Table 6.22 shows the maximum amount of memory consumed in different stages by our approach and the compared techniques. As the prototype approach utilizes MSA to generate a prototype, it takes much memory to align sequences/messages and to derive the prototypes. Our approach takes less amount of memory for LDAP and SOAP datasets compared to the prototype approach in the analysis phase. But, it takes more memory for Twitter and GoogleBooks because the responses are long in length and our approach infers format from the responses through identifying the keywords from the messages, which requires to store many intermediate values (*i.e.*, candidate keywords) in memory. At runtime, our approach takes less memory compared to both the whole cluster and prototype OSV as our approach neither searches for a closest matched interaction nor it infers any substitution rules at runtime (it utilizes the pre-inferred rules instead).

Table 6.22: Maximum Memory Usage (MB)

Dataset	No.	Memory Usage (MB)				
		OSV (Whole Cluster)	OSV (Prototype)		Our Approach	
			Analysis	Runtime	Analysis	Runtime
LDAP	19930	2821.69	10033.00	2356.45	592.09	207.78
SOAP	20000	4385.02	10832.30	2054.98	739.57	535.63
Twitter	1465	1394.03	5863.21	893.23	6033.18	351.84
GoogleBooks	1913	2295.66	10343.60	1534.29	11726.02	1840.33

6.5 Discussion

We have developed an approach to generating responses by considering the state of the service for the incoming requests. As described in Section 6.4.4, the evaluation results show that our approach is able to generate more accurate responses for both stateful and stateless service through inferring the service behavior model from the trace. Moreover, the format of the messages facilitates to improve the accuracy of the generated responses by preventing the generation of the malformed responses. At the same time, our approach is efficient enough in synthesizing responses for the incoming requests.

Our approach inferred the service behavior model to identify the dependency between messages and between message fields that allow us to keep track of the service state and to insert the appropriate payloads in the synthesized responses. The interactions are grouped into request-type based clusters to detect different types

of requests in the interaction trace after identifying the request type field in the request messages. The format for each request-type based messages is inferred to separate the message keywords from the payloads. The interactions are then further clustered based on the response type to reveal the different types of responses that are generated for the same request and for different requests after identifying the different types of responses per request-based cluster using the positional keyword-based clustering. In the same way, the format for each response-type based messages is inferred and stored in a map. The dependency between message fields (between request and the corresponding response messages) is inferred as substitution rules from the request and corresponding response formats and stored in another map. The interaction trace is partitioned and rearranged based on the *key payload*, with the aim of having the related interactions together. The message dependency model is inferred from the sequence of interactions to characterize the relationships among messages that in turn assists in keeping track of the service state. At runtime, an instance of the inferred model for each *key payload* associated with the request message is used to keep track of the service state for the *key payload* and select the type of response to send back in formulating responses for the incoming requests. Finally, the inferred data dependency model is used to determine and insert appropriate payloads in the synthesized responses.

6.5.1 Identical and Malformed Response Generation Result

The experimental results on four datasets both from stateless and stateful services demonstrate that our approach generates responses more accurately and efficiently compared to the existing approaches. The accuracy of the synthesized responses are measured based on the criteria described in Section 6.4.2, where *identical* is the highest degree of accuracy (the generated responses are exactly same as the expected) and *malformed* is the lowest (the generated responses are not interpretable). Figure 6.7 shows the result of generating identical (dark green) and malformed (dark red) responses by our approach and the compared approaches for *non-clean start*. It clearly shows that our approach outperforms the existing techniques in generating identical responses for stateful services (*i.e.*, LDAP and SOAP). The change of the service state is not observable in the interaction trace and the responses contain an extensive amount of the record-specific payloads (not seen in the messages) in the traces of stateless services (*i.e.*, Twitter and GoogleBooks). Thus, our approach generates less identical responses for stateless services. As Figure 6.7 shows, both the

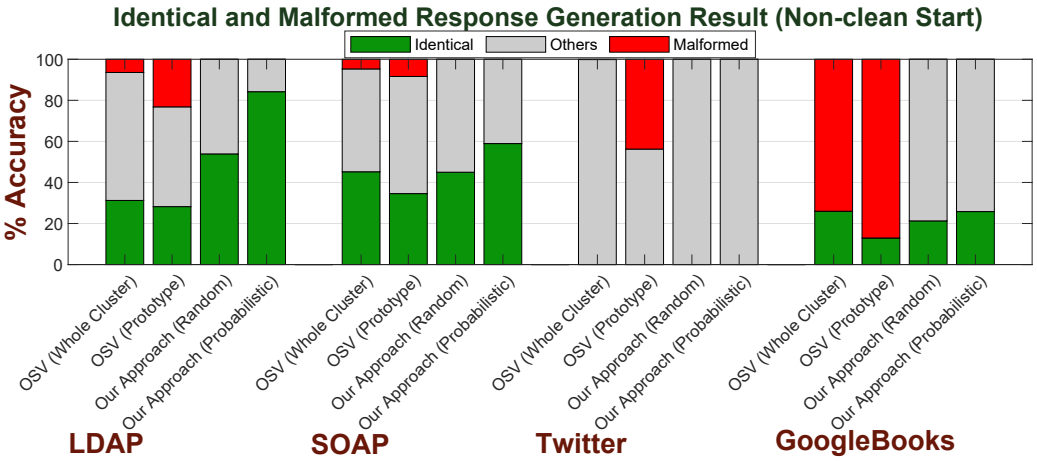


Figure 6.7: Identical and Malformed Response Generation Result (Non-clean Start)

whole cluster and prototype approaches generate malformed responses for stateless services (*i.e.*, Twitter and GoogleBooks), while our approach does not generate any malformed responses both for stateful and stateless services. It indicates our approach performs better compared to the existing techniques in generating responses both for stateful and stateless services and demonstrate the applicability of our approach both in stateful and stateless services.

6.5.2 Zoom-In Result (SOAP)

Table 6.23 shows the zoom-in result of response generation for the SOAP dataset to demonstrate the accuracy of our approach and the compared techniques for *each type of request message*. It shows that our approach generates identical responses for all incoming requests of `createNewAccount` and `closeAccount` requests as the responses for those requests do not contain payloads for non-key fields (*e.g.*, balance, account name, etc.). On the other hand, our approach generates data consistent responses (most) for the `getAccount` request as the responses of `getAccount` requests contain the current *balance* of the respective account. Thus, our approach is unable to generate the exact payloads for the *balance* field as we do not consider a "*data model*" in synthesizing responses. As Table 6.23 shows, none of the compared techniques (*i.e.*, the whole cluster and prototype OSV) is able to generate 100% identical responses for the `createNewAccount` and `closeAccount` requests. It also shows that our approach generates a number of protocol plausible responses for the `deposit` and `withdraw` requests and consequently, generates less number of identical responses compared to the existing approaches for those requests. This is so because the response (type) of a `withdraw` request not only depend on the

Table 6.23: Zoom-In Result of Response Generation (SOAP)

Request Type	No. of Interactions	Approach	Identical	Data Consistent	Protocol Exact	Protocol Plausible	Well-Formed	Malformed
Create-New-Account	2084	OSV (Whole Cluster)	66.99%	10.03%	7.53%	8.54%	0%	6.91%
		OSV (Prototype)	0%	29.66%	0%	31.14%	0%	39.20%
		Our Approach	100%	0%	0%	0%	0%	0%
Get-Account	5792	OSV (Whole Cluster)	8.66%	20.48%	0%	66.10%	0%	4.76%
		OSV (Prototype)	0.40%	87.62%	0%	11.98%	0%	0%
		Our Approach	12%	88%	0%	0%	0%	0%
Deposit	5631	OSV (Whole Cluster)	50.38%	5.09%	0%	40.45%	0%	4.08%
		OSV (Prototype)	57.04%	4.68%	0	38.28%	0%	0%
		Our Approach	34.37%	10.21%	0%	55.42%	0%	0%
Withdraw	5793	OSV (Whole Cluster)	49.40%	9.94%	0%	36.34%	0%	4.32%
		OSV (Prototype)	66.40%	8.56%	0%	25.04%	0%	0%
		Our Approach	57.32%	6.64%	0%	36.04%	0%	0%
Close-Account	700	OSV (Whole Cluster)	72%	3.86%	4.57%	19.58%	0%	0%
		OSV (Prototype)	0%	0%	0%	0%	0%	100%
		Our Approach	100%	0%	0%	0%	0%	0%

account number (key payload), also depends on the requested withdrawal amount and the current balance. The change of the service state due to the *non-key payload* leads to incorporate a misled transition in the inferred message dependency model. Thus, the message dependency model picks the wrong types of responses for the `withdraw` requests. Moreover, the selection of wrong response (type) for the `withdraw` request leads the wrong response (type) selection for the subsequent `deposit` requests. On the other hand, both the whole cluster and prototype-based OSV generate more identical responses compared to our approach for the `deposit` request, but the whole cluster approach also generates a few malformed responses. In the same way, the prototype-based OSV achieves comparatively higher accuracy

in generating identical response for the `withdraw` request. As described above, our approach is unable to track the service state if it is changed/updated due to the *non-key* fields. Overall, our approach achieves better accuracy compared to the whole cluster and prototype OSV in generating identical responses for the SOAP dataset.

6.5.3 Diversity of the Traces

The evaluation of our approach is performed on four datasets that are collected from real services. The datasets cover a range of application domains such as enterprise information management, SOAP-based banking services, REST-based information services, and social media applications. One of the threats to validity is the diversity of the datasets. However, our experimental datasets cover most of the services scenarios: i) *Stateful versus stateless* – A service can be either stateful or stateless. LDAP and SOAP datasets are collected from stateful services, while Twitter and GoogleBooks datasets from stateless services. ii) *A set of operations* – A service usually supports a set of requests and generates different types of response messages for each of these requests. Our experimental datasets LDAP, SOAP, Twitter, and GoogleBooks contain eight, five, six and two types of request messages respectively and consequently, each of them generates different types of response messages. iii) *Payloads* – The generated response usually contains a significant payload. The response messages for all datasets contain payload information especially the response messages of the Twitter and GoogleBooks datasets containing a huge amount of payloads. In general, the large majority of application services have the above characteristics.

6.5.4 Limitations

One of the limitations of our approach is its inability to automatic identify the key payload from the interactions on which the state of the service depends. In this work, the user has to provide a regular expression for identifying the key field and extracting the key payload information from the interactions. In future work, we plan to identify the key payload information automatically.

Another limitation of our approach is the lack of generating exact payloads for the non-key fields, *i.e.*, fields other than the key fields. Our approach does not consider an accurate *data model* to monitor the way of changing the data values of non-key fields over time and hence, generates inexact values for those fields while formulating responses. For example, in LDAP, the synthesized response of a `search`

request may contain outdated values in the mobile number field if the mobile number has been updated through invoking a `modify` request before the `search` request. To address this limitation, a data model should be inferred to store and keep track of the change history of payloads over time in addition to the service behavior model, to generate responses with exact payloads in the responses and model the service behavior more accurately.

Finally, our approach will only be applicable to textual protocols or services as the adapted process of identifying the request type and response type is particularly suitable for the traces of textual services. The service behavior model can still be useful to track the service state and to synthesize the responses for binary traces with the use of an alternative technique for identifying the request type and response type and extracting the request and response formats from the binary messages.

6.6 Summary

We have presented a new approach in service virtualization to create an emulation environment for stateful services. It extracts the dependency relationships among messages through mining the contextual information in the relevant interaction trace and come up with a service behavior model to track the service state while the client communicates with the virtual services. It exhibits that more accurate responses can be generated by considering the message dependency on the use of the behavior model. It also utilizes the message format in synthesizing responses to ensure message structure is retained in the synthesized responses and the dependency relationships between message fields utilized to insert appropriate payloads in the synthesized responses. Our approach has been able to achieve higher accuracy in generating responses, especially for stateful services. Moreover, it performs also well for stateless services.

Chapter 7

Conclusion and Future Work

In this thesis, we have introduced an approach and associated techniques to deriving "virtual" services from service interaction trace for stateful services. The derived virtual services allow testing of an interconnected system without requiring access to the connected services. In this chapter, we summarize the contributions of this thesis and outline some future works. Section 7.1 presents the key contributions of our research and we discuss directions for future work in Section 7.2.

7.1 Summary of Contributions

In this thesis we have demonstrated that our approach of generating responses can track the service state and mimic the behavior of stateful services by considering service state in synthesizing responses. The main objective of this thesis is to virtualize stateful services for providing a large and realistic testing environment. To achieve our objective, we have made the following key contributions:

Request Message Format: We have proposed an approach to identifying the request type field in the request messages and inferring the formats of the request messages. As a service commonly supports several types of request messages and thus, the request message contains a field to indicate its type. Based on this observation, our approach identifies the request type field from the request messages by analyzing the variable portions of the request messages. The request messages are partitioned into type-specific clusters using the identified request type field. A set of keywords is extracted for each request type-based messages by analyzing their frequency of occurrences independently in the messages. Finally, the format of the request messages is inferred for each request-type based message cluster by tokeniz-

ing the messages using the identified keywords and generalizing the keyword-payload patterns of individual messages. The experimental results have demonstrated that our approach achieves very high accuracy in clustering the messages based on the request type as it considers the variability (entropy) of the message fields and the dissimilarity among request messages in a cluster to identify the request type field. The results also have demonstrated that our approach infers the formats of the request messages precisely as it accurately extracts the type-specific keywords from the messages and generalizes the recurring keyword patterns across all messages in each of the message clusters.

Response Message Format: We have addressed the issue of inferring formats from the response messages with nested *repetition* structure. Moreover, the response messages do not always contain a message type field, rather, each type of response messages has its own set of keywords and has its own format. Existing techniques of inferring formats of the messages do not consider the repetitive sequences of keyword-payload in extracting keywords and thus, extract many false keywords, which leads to generating many more clusters than the actual clusters. On the other hand, our approach considers the position of the keywords and the frequency of messages containing a keyword rather than the number of times a keyword appears, which leads to the accurate identification of keywords. The extracted keywords are further refined based on the variations of positions, *i.e.*, the standard deviation of keywords' positions. Each response message is then converted into a weighted vector based on the positional keywords appearing in the message. Finally, response messages are clustered based on the dissimilarity among messages (*i.e.*, weighted vector) and formats of response messages are inferred. The nested repetition structure of the response message is effectively handled by our approach of inferring formats as it generalizes the keyword-payload patterns across all response messages of a response-based cluster. The experimental results have demonstrated that our approach accurately clusters the response messages of request type-based clusters, and infers formats of the response messages precisely due to the capability of dealing with the *repetitive* keyword-payload patterns.

Behavior Model and Response Generation: We have proposed a novel technique to synthesize response messages for incoming request messages by considering the service state. It identifies the dependency among messages (*i.e.*, control dependency) by analyzing and extracting the contextual information from the interaction traces. A sequence of request-response type pair is generated for each key payload to capture the message sequence patterns (key payload specific), which are responsible to generate different types of response messages for different types of requests

and even for the same requests. The dependency among messages is inferred by incorporating the message sequence patterns for all key payloads. Moreover, the dependencies between message fields (*i.e.*, data dependency) are extracted from the interaction trace to determine appropriate payloads in the synthesized responses. Using the inferred service behavior model, our technique synthesizes the responses by determining the response type to send from the control dependency model and including appropriate payloads from the data dependency model. The proposed technique is evaluated for both stateful and stateless services and the experimental results have shown significant improvements in generating identical responses for stateful services over existing techniques. The results also have demonstrated that our approach greatly improves the accuracy in synthesizing responses for stateless service by preventing the generation of malformed responses. Moreover, our approach generates responses more efficiently compared to the existing techniques for both stateful and stateless services and hence, can provide a comprehensive testing environment for the system under test in service virtualization.

7.2 Future Work

In this dissertation, we have proposed techniques to derive service models for stateful services so that they can be used in emulating the behavior of the dependent services. A number of research works can be done to further improve the quality of the derived service models for representing the behavior of the actual services. We outline some of these potential future research directions below.

7.2.1 Key Payload Identification

In our current response generation approach, the interactions are partitioned based on the *key payload* for having the relevant interactions together. As described in Section 6.2.1, the service state is independent for each key payload and the dependency among messages is considered uniquely for each key payload associated with the interactions. Therefore, the interactions are partitioned based on the key payloads for identifying the dependency between messages. However, we assume that the key payload is identified by a user (given regular expression). To automate this key payload identification, we intend to investigate the process of identifying such key payloads automatically from the service interaction trace. One probable way is to analyze the interactions in-between two same types of requests that are generated different types of response messages. Some of the message fields may be

excluded from further consideration based on the semantic analysis. The ability to automatically identify the key payload field from the interaction trace will remove the required human efforts to identify such fields.

7.2.2 The Impact of Non-key Payloads

We have developed an approach to identify the dependency between messages to keep track of the service state and synthesize responses for stateful services based on the service state. We have analyzed the traces based on the key payloads and identified the situation of generating different types of responses for the same types of request messages. However, the service state can also be changed due to non-key fields. Such effects of non-key payloads are ignored in our current approach and hence, protocol-plausible responses are generated in such circumstances. For example, in a banking service, the response may generate different responses for a `withdraw` request if the account does not have sufficient balance. Our approach identifies these two different types of responses that are are generated for `withdraw` request, but didn't identify the reason for generating such response messages. One probable way is to further analyze the payloads and identify such interplay between the key and non-key payloads in determining the responses. The ability to identify such relationships with non-key payloads will reduce the generation of protocol-plausible response messages.

7.2.3 Data Model Inference

In this research, our approach of synthesizing responses has extracted the dependency among message fields to determine the appropriate payloads in the synthesized responses. In our current approach, the data dependency model identifies the relationships of the message fields with the corresponding request message and/or the preceding request message. But, some of the payloads can be changed over time and such changes in the payloads are not considered in the data dependency model. Therefore, our approach synthesizes responses with inexact payloads in such fields (*i.e.*, data-consistent responses) although it identifies the types of responses accurately. For example, in a banking service, the balance of an account changes due to `withdraw` or `deposit` requests. Similarly, in the CA IM service, the name, mobile number, address, etc. of a record (*i.e.*, employee) can be changed/updated using a `modify` request. Thus, the response message for a request following such a `modify` request should contain the updated/modified payloads for those modified fields. A *data model* may be inferred from the traces to monitor such changes in the payloads.

The ability to infer such a data model may be useful in synthesizing responses with exact payloads and consequently, to further enhance the quality of emulation.

7.2.4 Inter-service Interactions

In this dissertation, we have investigated the change of the service state due to the preceding sequence of interactions, and considered the service state in synthesizing responses. But, some dependencies between services may exist and the service state of a connected or dependent service may be changed due to the interactions with other services. For example, in banking service, an account can be authorized for direct debit and then, the authorized organization can take payments (*i.e.*, `withdraw`) automatically from the account. Such interactions with other services change the state of the respective account. Thus, the derived service models need to consider such interactions between services, to handle multiple interrelated services and consequently, to represent the behavior of the dependent services more accurately. In this dissertation, we have considered the effect of other services as black-box, *i.e.*, the traces are collected by intercepting the communication of a single service but the collected traces encapsulate the effect of other services. The transformation of the messages between services are not explored explicitly but the messages by the service in focus encapsulate the effect of the dependent services. In future work, we plan to investigate the dependency between services. One probable way is to correlate the *timestamps* of the messages from multiple traces for identifying the possible dependency relationships among themselves. The identification of such correlations between services will be assisting in generating more accurate responses to the incoming request messages.

Appendices

Appendix A

Zoom-In Results

A.1 Request Format Extraction Result

Table A.1 shows the Zoom-In results of format extraction from the request messages. We evaluate the inferred format for each request-type based cluster and report the results, *i.e.*, precision, recall and F-measure in Table A.1. As Table A.1 shows, all three approaches achieve 100% precision and recall for the LDAP **`Bind`** messages. This is because the payloads in **`Bind`** messages do not vary from messages to messages, *i.e.*, the same **`Bind`** request is used for authentication and the interaction trace contains the same **`Bind`** request message 100 times as we collect LDAP trace from 100 sessions. In the same way, ProDecoder and our approach achieve 100% precision and recall for **`Unbind`** messages. However, AutoReEngine achieves lower accuracy for **`Unbind`**. Unlike to **`Bind`** messages, **`Unbind`** messages contain different payloads for the "**`message id`**" field. AutoReEngine puts most of the **`Unbind`** messages into the clusters of other request types and creates the **`Unbind`** cluster with very few **`Unbind`** messages (about 10% only), which leads the inferred format containing payload and thus, achieves very low precision and recall.

AutoReEngine creates a separate cluster for each **`ModifyDN`** messages, which leads to infer too specific format and achieves 0.00 in both precision and recall. On the other hand, ProDecoder creates mix cluster for **`ModifyDN`** and hence, achieves very low precision. Both ProDecoder and AutoReEngine create mixed cluster for the rest of the request types and thus, achieve low precision and recall. For the mixed clusters, the precision and recall of the inferred format depend on the nature of the mixing: i) if the mixed cluster contains messages of two different types of requests and one of the request type dictates the mixed cluster, precision will be relatively

high, ii) if the mixed cluster contains messages of multiple clusters (more than two clusters) and none of the message type alone dictates the mixed cluster, the inferred format will be imprecise and achieve lower precision. For example, the precision of `Modify` cluster for ProDecoder is 0.90 as it creates the `Modify` cluster accurately for most of the folds and `Modify` dictates the cluster (about 90% of the messages are of the `Modify` type) when it mixes with `Bind` messages. On the other hand, the precision of the `ModifyDN` cluster for ProDecoder is 0.08 as it puts `ModifyDN` messages to other cluster and mixed the rest with the `Compare` messages. In contrast, our approach classifies messages accurately and hence, infer more accurate formats through identifying the type-specific keywords and by generalizing the keyword sequences across messages in a cluster.

For SOAP, both ProDecoder and AutoReEngine create 4 clusters and infer 4 formats instead of 5 ground-truth clusters/formats. AutoReEngine puts the messages of `closeAccount` into the `createNewAccount` cluster and hence, achieves low precision for `createNewAccount`. However, it accurately classify the request messages of other types and thus, achieves high precision and recall. On the other hand, ProDecoder puts the messages of `closeAccount` into all clusters and consequently, infers imprecise formats for all types and delivers low precision.

For Twitter, both ProDecoder and AutoReEngine create mixed cluster for `statusesupdate`. AutoReEngine puts all messages of `statusesupdate` & `statusesuser_timelinejsonuser_id` into one cluster and thus, infer imprecise format and achieve 0.00 in both precision and recall. While ProDecoder mixes the message of `statusesupdate`, `statusesuser_timelinejsonuser_id` & `statusesuser_timelinejsonscreen_name` and thus, achieve 0.00 as well. AutoReEngine accurately cluster the message of `statusesuser_timeline- jsonscreen_name` and hence, achieve high precision and recall in format extraction from that type of messages. ProDecoder generates mix cluster for `statusesuser _timelinejsonscreen_name` and hence, infer imprecise format for that cluster. It accurately cluster the messages of `statusesshowjsonid` and infer format precisely.

For GoogleBooks, AutoReEngine mixes all messages into one cluster and thus, achieve lower precision in format extraction. ProDecoder clusters the messages accurately for most of the folds and generates mix cluster for a few folds and hence, achieve relatively higher accuracy than AutoReEngine in extracting request format from the messages.

In general, our approach achieves high accuracy in format extraction because of two reasons: i) it clusters the request messages accurately based on the request-type after identifying the request type field through entropy analysis and by consider-

ing the dissimilarity among messages in a cluster, and ii) accurately identifying type-specific keywords and removing false keywords (substrings) through frequency subtraction and inferring format through generalization, which allows the inferred format accept yet unseen, valid messages.

Table A.1: Request Format Extraction Result (Zoom-In)

Dataset	**Cluster**	**ProDecoder**			**AutoReEngine**			**Our Approach**		
		P	R	F	P	R	F	P	R	F
LDAP	Bind	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Add	0.40	1.00	0.57	0.60	0.19	0.29	1.00	1.00	1.00
	Delete	0.84	1.00	0.91	0.26	0.01	0.02	1.00	1.00	1.00
	Search	1.00	0.49	0.66	0.84	0.55	0.66	1.00	1.00	1.00
	Modify	0.90	1.00	0.95	0.50	0.03	0.06	1.00	1.00	1.00
	ModifyDN	0.08	1.00	0.15	0.00	0.00	0.00	1.00	1.00	1.00
	Compare	0.65	0.64	0.64	0.94	0.17	0.29	1.00	1.00	1.00
	Unbind	1.00	1.00	1.00	0.60	0.10	0.17	1.00	1.00	1.00
	Weighted Score	0.76	0.83	0.79	0.63	0.21	0.31	1.00	1.00	1.00
SOAP	createNewAccount	0.10	1.00	0.18	0.77	1.00	0.87	1.00	1.00	1.00
	deposit	0.28	1.00	0.44	1.00	1.00	1.00	1.00	1.00	1.00
	getAccount	0.28	1.00	0.44	1.00	1.00	1.00	1.00	1.00	1.00
	withdraw	0.29	1.00	0.45	1.00	1.00	1.00	1.00	1.00	1.00
	closeAccount	-	-	-	-	-	-	1.00	1.00	1.00
	Weighted Score	0.26	1.00	0.41	0.97	1.00	0.99	1.00	1.00	1.00
Twitter	friendshipsshow	0.80	0.80	0.80	0.62	1.00	0.77	1.00	1.00	1.00
	statusesupdate	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00

Table A.1 Continued: Request Format Extraction Result (Zoom-In)

Dataset	Cluster	ProDecoder			AutoReEngine			Our Approach		
		P	R	F	P	R	F	P	R	F
	statusesuser_timelinejsonuser_id	0.37	1.00	0.54	-	-	-	1.00	1.00	1.00
	statusesuser_timelinejsonscreen_name	0.50	1.00	0.67	1.00	1.00	1.00	1.00	1.00	1.00
	statusesshow	1.00	1.00	1.00	-	-	-	1.00	1.00	1.00
	searchtweets	0.80	0.80	0.80	1.00	0.98	0.99	1.00	1.00	1.00
	Weighted Score	0.77	0.77	0.77	0.78	0.91	0.84	1.00	1.00	1.00
Google-Books	search_volume	0.89	0.97	0.93	0.74	1.00	0.85	1.00	1.00	1.00
	search_bookshelf	0.89	1.00	0.94	-	-	-	1.00	1.00	1.00
	Weighted Score	0.89	0.98	0.93	0.74	1.00	0.85	1.00	1.00	1.00

A.2 Response Message Clustering Result

Table A.2 shows the Zoom-In result of clustering the responses of each request-based cluster. As it shows, all three approaches cluster the `Bind` and `Unbind` responses accurately for the LDAP dataset as the responses of the `Bind` requests are same and responses for the `Unbind` requests are *empty*. For the rest of the request-based clusters of LDAP, both ProDecoder and AutoReEngine cluster responses inaccurately due to the misidentification of keywords from the messages. On the other hand, our approach extracts keywords from the messages by considering the keyword's positions in the messages and hence, achieves 100% accuracy in clustering the response messages of all datasets. For SOAP, all three approaches cluster the responses of `closeAccount` request accurately as the responses do not vary from the responses to responses of the `closeAccount` request. Again, for the rest of the request-based clusters, both ProDecoder and AutoReEngine mixed the responses of multiple clusters into one. For Twitter, ProDecoder cluster the responses accurately except

for the `statusesshow` and `searchtweets` clusters as those clusters contain the messages of one type of responses only. It creates mixed cluster for the responses of the `statusesshow` and `searchtweets` clusters. On the other hand, AutoReEngine creates many more clusters than the actual except for the `statusesshow` and `searchtweets` clusters. For example, it creates 22 clusters for the responses of the `statusesupdate`-based cluster. For the GoogleBooks dataset, all three approaches cluster responses of the `search_bookshelf` cluster. This is because the `search_bookshelf` request either returns an error response when the searched bookshelf is not found in the database or the detail information about a bookshelf. Those responses contain a distinctive set of keywords and thus, all three approaches cluster such messages using the extracted keywords. But, both ProDecoder and AutoReEngine are unable to cluster the responses accurately of the `search_volume` cluster. In general, our approach considers the position of the keywords in the messages of *repetitive* sequence of keyword-payload and hence, it extracts the keywords accurately from the messages and achieves higher accuracy in clustering the responses.

Table A.2: Response Clustering Result (Zoom-In)

Dataset	**Cluster**	**ProDecoder**			**AutoReEngine**			**Our Approach**		
		P	R	F	P	R	F	P	R	F
LDAP	Bind	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Add	1.00	1.00	1.00	0.15	0.19	0.17	1.00	1.00	1.00
	Delete	1.00	0.69	0.81	0.15	0.26	0.19	1.00	1.00	1.00
	Search	0.28	0.22	0.25	0.27	0.21	0.24	1.00	1.00	1.00
	Modify	0.42	0.46	0.44	0.44	0.58	0.50	1.00	1.00	1.00
	ModifyDN	0.70	0.75	0.72	0.89	0.87	0.88	1.00	1.00	1.00
	Compare	1.00	0.85	0.92	0.32	0.38	0.35	1.00	1.00	1.00
	Unbind	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Weighted Score	0.73	0.65	0.69	0.30	0.36	0.33	1.00	1.00	1.00
SOAP	createNewAccount	0.26	0.14	0.18	1.00	1.00	1.00	1.00	1.00	1.00
	deposit	0.66	0.75	0.70	1.00	1.00	1.00	1.00	1.00	1.00
	getAccount	0.34	0.38	0.36	1.00	0.39	0.56	1.00	1.00	1.00

Table A.2 Continued: Response Clustering Result (Zoom-In)

Dataset	Cluster	ProDecoder			AutoReEngine			Our Approach		
		P	R	F	P	R	F	P	R	F
	withdraw	0.54	0.55	0.54	1.00	0.78	0.88	1.00	1.00	1.00
	closeAccount	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Weighted Score	0.50	0.52	0.51	1.00	0.77	0.87	1.00	1.00	1.00
Twitter	friendshipsshow	1.00	1.00	1.00	1.00	0.10	0.18	1.00	1.00	1.00
	statusesupdate	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00
	statusesuser _timelinej-sonuser_id	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00
	statusesuser _timelinejson-screen_name	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00
	statusesshow	0.01	0.01	0.01	0.75	0.09	0.15	1.00	1.00	1.00
	searchtweets	0.53	0.60	0.56	0.69	0.16	0.27	1.00	1.00	1.00
	Weighted Score	0.62	0.65	0.63	0.82	0.11	0.20	1.00	1.00	1.00
Google-Books	search_volume	0.04	0.02	0.03	1.00	0.07	0.13	1.00	1.00	1.00
	search_book-shelf	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Weighted Score	0.29	0.27	0.28	1.00	0.31	0.48	1.00	1.00	1.00

A.3 Response Format Extraction Result

Table A.3 shows Zoom-In result of inferring formats of the response messages. As Table A.3 shows, all three approaches achieve 100% accuracy in format inference for the responses of the LDAP `Bind` and `Unbind` clusters. The payloads of the `Bind` responses do not vary from responses to responses, while `Unbind` responses are empty. ProDecoder creates a mixed cluster for other types of the LDAP messages and

hence, achieves lower precision and recall for the inferred formats of the response messages. On the other hand, AutoReEngine achieves 100% precision and recall for some types (*e.g.*, `Add`) even though it creates more clusters than the actual. This is because we evaluate the response in "OR" -ed fashion, *i.e.*, the inferred formats are considered valid if any one of them accepts the message of that cluster. Due to the same reason, it achieves 100% accuracy in format extraction for some types of Twitter messages. For SOAP, ProDecoder achieves 100% accuracy in format extraction for the responses of the `closeAccount` request as it correctly clusters the messages. But, it achieves lower accuracy for all other types. AutoReEngine is unable to achieve 100% accuracy for any type of SOAP messages. In contrast, our approach accurately infers formats of the responses for the SOAP dataset. For Twitter, ProDecoder achieves 100% accuracy for the `statusesshow` and `statusesupdate` clusters as the responses of those clusters contain a single type of responses and those responses do not contain *repetition* sequence of keyword-payload. Due to the same reason, both ProDecoder and AutoReEngine achieve 100% accuracy in format extraction for the responses of `search_bookshelf` in GoogleBooks dataset. As Table A.3 shows, the accuracy of the `statusesuser_timelinejsonuser_id` and `statusesuser_timelinejsonscreen_name` (Twitter) for ProDecoder is lower even though it accurately clusters the responses of those types. This is due to the incapability of generalizing the individual message pattern into the inferred format. The responses of `search_volume` cluster in GoogleBooks also contain the *repetition* sequence of keyword-payload and hence, both ProDecoder and AutoReEngine achieve lower accuracy, while our approach achieves 100% accuracy in extracting formats of such response messages.

Table A.3: Response Format Extraction Result (Zoom-In)

Dataset	**Cluster**	**ProDecoder**			**AutoReEngine**			**Our Approach**		
		P	R	F	P	R	F	P	R	F
LDAP	Bind	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Add	0.67	1.00	0.80	1.00	1.00	1.00	1.00	1.00	1.00
	Delete	0.74	0.81	0.77	1.00	1.00	1.00	1.00	1.00	1.00
	Search	0.99	0.94	0.96	0.71	0.64	0.67	1.00	1.00	1.00
	Modify	0.80	0.96	0.87	0.50	0.87	0.63	1.00	1.00	1.00
	ModifyDN	0.88	0.94	0.90	0.49	0.81	0.61	1.00	1.00	1.00
	Compare	0.82	0.81	0.81	1.00	1.00	1.00	1.00	1.00	1.00

Table A.3 Continued: Response Format Extraction Result (Zoom-In)

Dataset	Cluster	ProDecoder			AutoReEngine			Our Approach		
		P	R	F	P	R	F	P	R	F
	Unbind	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Weighted Score	0.81	0.91	0.86	0.83	0.89	0.86	1.00	1.00	1.00
SOAP	createNewAccount	0.93	1.00	0.96	0.93	1.00	0.96	1.00	1.00	1.00
	deposit	0.80	0.93	0.86	0.81	0.93	0.87	1.00	1.00	1.00
	getAccount	0.85	1.00	0.92	0.79	1.00	0.88	1.00	1.00	1.00
	withdraw	0.75	0.73	0.74	0.56	0.60	0.58	1.00	1.00	1.00
	closeAccount	1.00	1.00	1.00	0.50	0.50	0.50	1.00	1.00	1.00
	Weighted Score	0.82	0.90	0.86	0.73	0.84	0.78	1.00	1.00	1.00
Twitter	friendshipsshow	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	statusesupdate	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	statusesuser_timelinejsonuser_id	1.00	0.83	0.91	1.00	0.80	0.89	1.00	1.00	1.00
	statusesuser_timelinejsonscreen_name	1.00	0.88	0.94	1.00	0.79	0.88	1.00	1.00	1.00
	statusesshow	0.61	0.74	0.67	0.78	0.81	0.79	1.00	0.91	0.95
	searchtweets	0.63	0.87	0.73	0.68	0.99	0.81	1.00	1.00	1.00
	Weighted Score	0.77	0.89	0.83	0.82	0.95	0.88	1.00	0.99	0.99
Google-Books	search_volume	0.98	0.74	0.84	0.95	0.90	0.92	1.00	1.00	1.00
	search_bookshelf	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Weighted Score	0.99	0.81	0.89	0.96	0.93	0.94	1.00	1.00	1.00

A.4 Response Format Inference Time

Table A.4 shows the Zoom-In result of required time for clustering the responses of each request-based cluster.

Table A.4: Time (s) for Clustering and Inferring Response Formats (Zoom-In)

Dataset	Cluster	ProDecoder		AutoReEngine		Our Approach	
		C	F	C	F	C	F
LDAP	Bind	1.38	0.03	0.96	0.84	0.14	0.96
	Add	14.15	1.97	0.99	57.20	23.64	3.33
	Delete	17.87	2.53	1.32	75.17	44.10	4.27
	Search	31.79	3.73	2.05	209.72	63.45	6.43
	Modify	22.45	1.30	1.52	103.70	52.08	7.81
	ModifyDN	2.10	0.04	0.96	0.98	0.25	4.98
	Compare	18.92	1.60	1.66	75.12	28.50	4.98
	Unbind	0.01	0.00	0.00	0.00	0.00	0.02
	Avg. Time (message-weighted)	20.12	2.11	1.47	100.33	40.32	5.21
SOAP	createNewAccount	21.25	1.52	15.17	10.54	90.85	23.86
	deposit	44.83	4.10	20.70	24.88	90.62	42.54
	getAccount	39.73	5.36	19.24	48.14	60.15	40.70
	withdraw	44.44	4.06	21.55	24.54	40.50	43.13
	closeAccount	5.04	0.14	3.93	0.71	10.32	16.44
	Avg. Time (message-weighted)	39.12	3.99	19.30	28.67	65.04	39.07
Twitter	friendshipsshow	22.39	5.45	2.71	40.69	11.60	1.15
	statusesupdate	26.13	2.54	20.78	4.15	14.96	228.63

Table A.4 Continued: Time (s) for Clustering and Inferring Response Formats (Zoom-In)

Dataset	Cluster	ProDecoder		AutoReEngine		Our Approach	
		C	F	C	F	C	F
	statusesuser_timelinejsonuser_id	308.43	67.78	68.13	243.21	0.78	73.20
	statusesuser_timelinejsonscreen_name	306.25	67.68	69.60	214.41	0.71	71.76
	statusesshow	115.78	25.26	11.27	42.93	0.33	12.90
	searchtweets	3185.11	4282.58	388.65	2617.80	5.53	586.89
	Avg. Time (message-weighted)	1403.55	1839.24	173.31	1148.57	6.74	271.21
Google-Books	search_volume	2575.88	57.11	1091.57	1794.13	37.24	0.67
	search_bookshelf	1.88	0.04	2.89	0.04	9.73	6.58
	Avg. Time (message-weighted)	1907.15	42.28	808.73	1328.02	30.09	2.20

A.5 Inferred Service Behavior

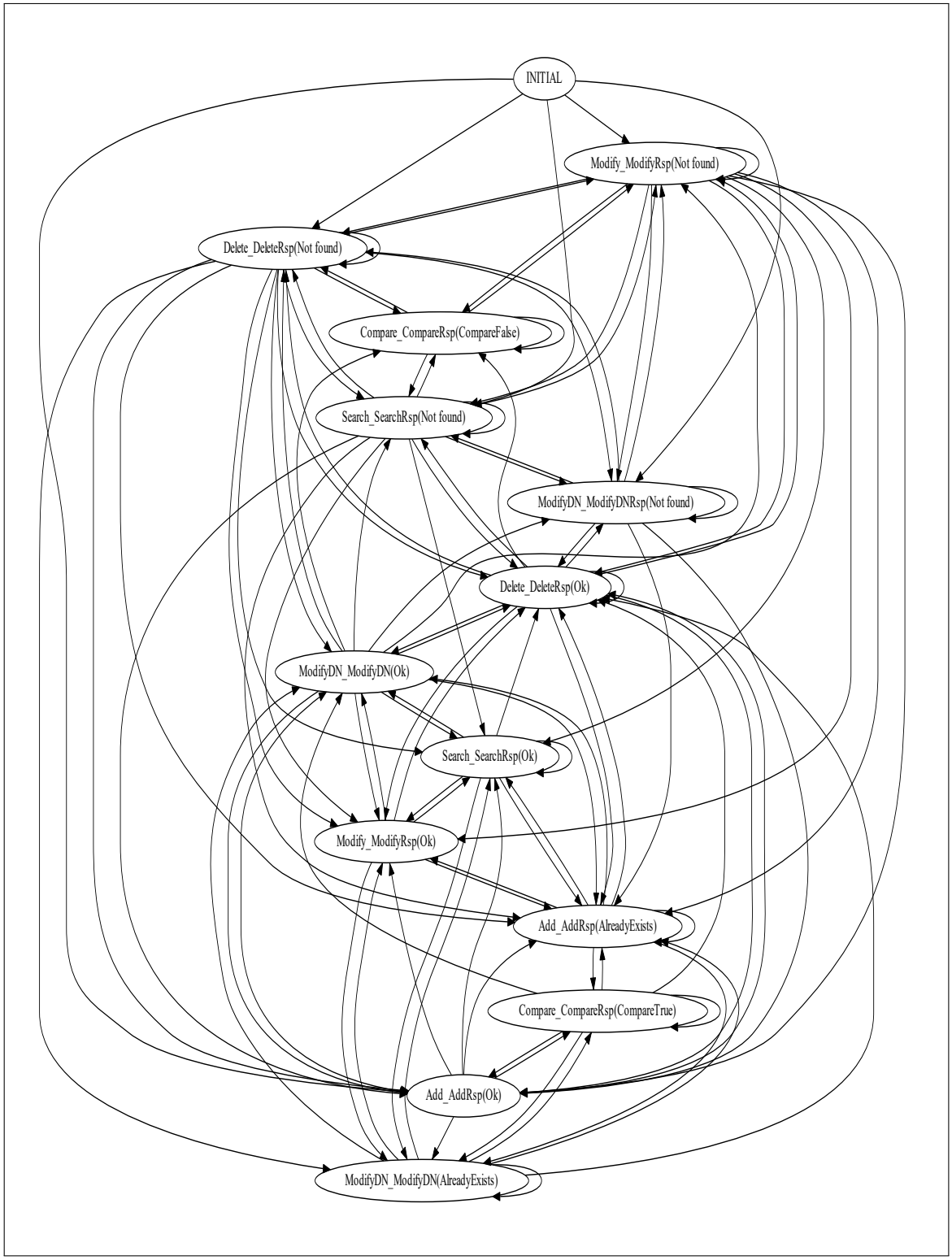


Figure A.1: Inferred Model from LDAP Trace (Clean start)

A.6 Generation of Protocol Plausible Responses

This section presents the details of generating protocol plausible responses by our approach. In general, our approach keeps track of the service state and generates responses based on the current state of the services, therefore, it is expected that our approach generates at least a data consistent response. Unfortunately, our approach generates a few protocol plausible responses, especially when the service state is changed due to *non-key* fields or the value of *key* field is updated during communication. We present the detail process of protocol plausible response generation with an example for LDAP dataset.

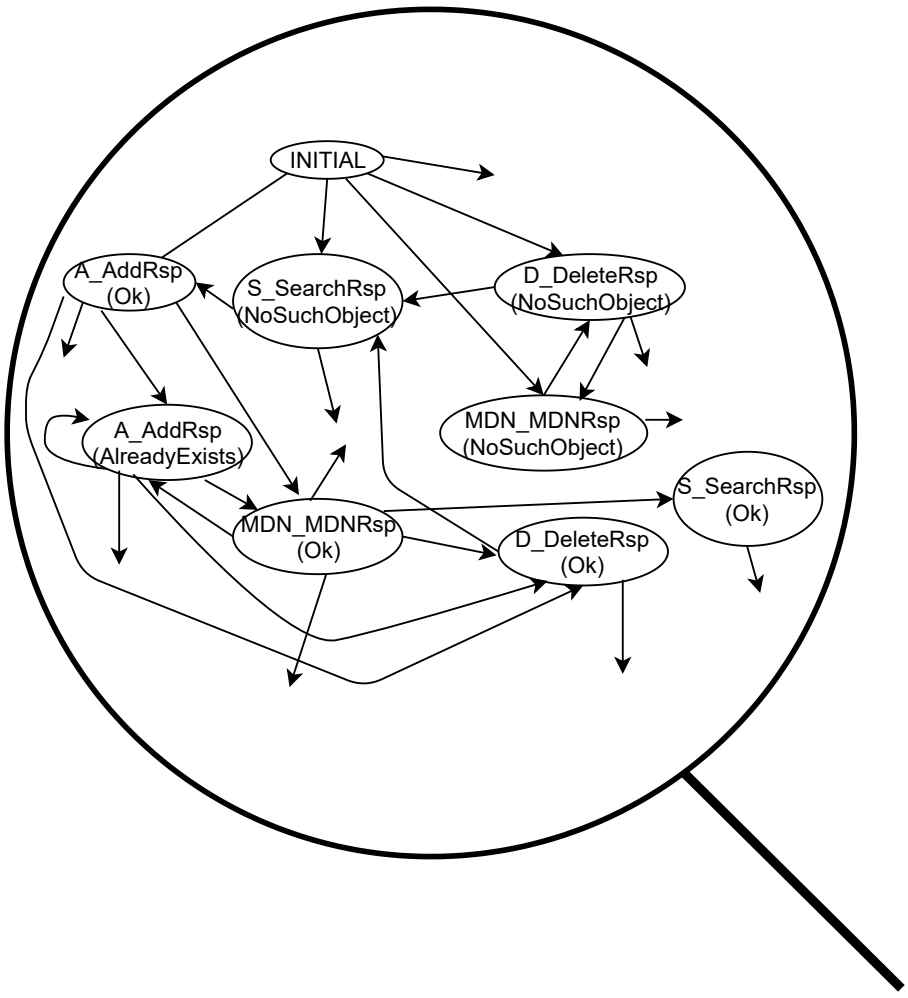


Figure A.2: Partial Service Behavior Model

The service behavior model inferred by our approach as depicted in Figure A.1 contains so many nodes & edges, we use a partial model shows in Figure A.2 to describe the details of generating protocol plausible responses by our approach.

Let us consider the interaction trace shows in Table A.5 as an example. At

Table A.5: Incoming Requests and Expected Responses

Index	Incoming Request	Expected Response
1	{id:1,op:S,cn:Judith}	{id:1,op:SearchRsp,result:Not Found}
2	{id:2,op:A,cn:Judith}	{id:2,op:AddRsp,result:Ok}
3	{id:3,op:MDN,Oldcn:Judith, Newcn:Gavin}	{id:3,op:MDNRsp,result:Ok}
4	{id:4,op:A,cn:Gavin}	{id:4,op:AddRsp,result:Already Exists}
5	{id:5,op:D,cn:Gavin}	{id:5,op:DeleteRsp,result:Ok}

runtime, that is, at the time of generating responses for the incoming requests, our approach create an instance of the inferred model for each different key records associated in the request messages as described in Section 6.2.4. Therefore, the service state for the record "**Judith**" is set to "**INITIAL**" as described in the Algorithm 5 in Chapter 6. Then it selects "**S_NoSuchObject**" as the response type for the first request of Table A.5 and generates the following response after transformation, which is an *identical* response as per the evaluation criteria described in Section 6.4.2

```
{id:1,op:SearchRsp,result:Not Found}
```

Similarly, it generates *identical* response for the second and third requests in Table A.5. But, the value of the key-record (*i.e.*`cn`) is changed after executing the third request, that is, `ModifyDN`) request takes two common names and the old common name is replaced by a new common name. In third request (index:3), the value of `cn` (*i.e.*`Judith`) is replaced by `Gavin`. Therefore, the key record `Judith` is no more available in the data store and the key record `Gavin` retains the current state of the service for the record `Judith`. But, our approach assigns "**INITIAL**" as the current state when the key record appears for first time and it generates the following response for the fourth (index:4) request

```
{id:4,op:AddRsp,result:ok}
```

The above generated response is evaluated as *protocol-plausible* based on the evaluation criteria described in Section 6.4.2. Similarly, our approach generates *protocol-plausible* response for the fifth (index:5) request in Table A.5 and it continues to generate *protocol-plausible* responses for the record `Gavin` until an `add` request becomes observable in the interaction trace.

Appendix B

Interaction Trace Example

B.1 Lightweight Directory Access Protocol (LDAP)

The LDAP example trace contains eight types of request messages, which are **`Add`**, **`Delete`**, **`Bind`**, **`Search`**, **`Modify DN`**, **`Modify`**, **`Compare`**, and **`Unbind`**. For each type of request messages, we present an example request message and the corresponding response message.

B.1.1 Bind Operation

The **`Bind`** operation is used to authenticate a client to the LDAP server. An example of such a request and the corresponding response messages are following

LDAP Bind request

```
LDAP Bind Request Message ID: 1 LDAP Bind Request Protocol Op LDAP Version: 3
Bind dn: cn=admin,dc=ca,dc=com Authentication Data: Authentication Type:
Simple Bind Password: 1228013670
```

LDAP Bind response

```
LDAP Bind Response Message ID: 1 LDAP Bind Response Protocol Op Result
Code: 0 (Success)
```

B.1.2 Add Operation

The **Add** operation is used to create a new entry (*i.e.*, record) into the LDAP server. An example of such a request and the corresponding response messages are following

LDAP Add request

```
LDAP Add Request Message ID: 10 LDAP Add Request Protocol Op dn: cn=Susana GIDDINGS
,ou=Records,ou=Customer,dc=ca,dc=com mail: Susana.GIDDINGS@ca.com mobile: 5530146
description: Customer Service objectClass: inetOrgPerson title: Customer Service
sn: GIDDINGS cn: Susana GIDDINGS
```

LDAP Add response

```
LDAP Add Response Message ID: 10 LDAP Add Response Protocol Op Result
Code: 0 (Success)
```

B.1.3 Search Operation

The **Search** operation is used to perform a search in the LDAP directory server. An example of such a request and the corresponding response messages are following

LDAP Search request

```
LDAP Search Request Message ID: 24 LDAP Search Request Protocol Op Base DN:
cn=Judith GIDDINGS,ou=Administration,ou=Corporate,dc=ca,dc=com Scope: 0 (baseObject)
Deref Aliases: 3 (derefAlways) Size Limit: 1 Time Limit: 0 Types Only: false Filter:
(cn=Judith STONE,ou=Management,ou=Corporate) Attributes: 1.1
```

LDAP Search response

```
LDAP Search Result Done Message ID: 24 LDAP Search Result Done Protocol Op Result
Code: 32 (No Such Object) Matched DN: ou=Administration,ou=Corporate, dc=ca,dc=com
```

B.1.4 Delete Operation

The **Delete** operation is used to delete an existing entry (*i.e.*, record) from the LDAP directory server. An example of such a request and the corresponding response messages are following

LDAP Delete request

```
LDAP Delete Request Message ID: 4 LDAP Delete Request Protocol Op dn: cn=Gavin
GIDDINGS,ou=Records,ou=Customer,dc=ca,dc=com
```

LDAP Delete response

```
LDAP Delete Response Message ID: 4 LDAP Delete Response Protocol Op Result
Code: 0 (Success)
```

B.1.5 Modify Operation

The `Modify` operation is used to update/modify an existing entry (*i.e.*, record) in the LDAP directory server. An example of such a request and the corresponding response messages are following

LDAP Modify request

```
LDAP Modify Request Message ID: 10 LDAP Modify Request Protocol Op dn:
cn=Mark MAJOR,ou=Management,ou=Corporate,dc=ca,dc=com changetype: modify
add: telephoneNumber telephoneNumber: 6176185 replace: description
description: Acting Supervisor delete: mobile
```

LDAP Modify response

```
LDAP Modify Response Message ID: 10 LDAP Modify Response Protocol Op Result Code:
16 (No Such Attribute) Error Message: modify/delete: mobile: no such attribute
```

B.1.6 Modify DN Operation

The `Modify DN` operation is used to update/modify *DN* of an existing entry (*i.e.*, record) in the LDAP directory server. An example of such a request and the corresponding response messages are following

LDAP Modify DN request

```
LDAP Modify DN Request Message ID: 42 LDAP Modify DN Request Protocol Op Current
Entry DN: cn=Judith STONE,ou=Administration,ou=Corporate,dc=ca,dc=com New RDN:
cn=Mark GIDDINGS Delete Old RDN: true New Superior: ou=Records, ou=Customer,dc=
ca,dc=com
```

LDAP Modify DN response

```
LDAP Modify DN Response Message ID: 42 LDAP Modify DN Response Protocol Op Result
Code: 0 (Success)
```

B.1.7 Compare Operation

The **Compare** operation is used to check whether an existing entry (*i.e.*, record) in the LDAP directory server has the specified attributes. An example of such a request and the corresponding response messages are following

LDAP Compare request

```
LDAP Compare Request Message ID: 16 LDAP Compare Request Protocol Op Entry
DN: cn=Susana RAYMOND,ou=Infrastructure,ou=Support,dc=ca,dc=com Attribute
Type: mail Assertion Value: Susana.MAJOR@ca.com
```

LDAP Compare response

```
LDAP Compare Response Message ID: 16 LDAP Compare Response Protocol Op Result
Code: 32 (No Such Object) Matched DN: ou=Infrastructure,ou=Support, dc=ca,dc=com
```

B.1.8 Unbind Operation

The **Unbind** operation is used to disconnect from the LDAP server. An example of such a request and the corresponding response messages are following

LDAP Unbind request

```
LDAP Unbind Request Message ID: 39 LDAP Unbind Request Protocol Op
```

LDAP Unbind response

```
""
```

B.2 Simple Object Access Protocol (SOAP)

The SOAP example trace contains five types of request messages, which are **CreateNewAccount, GetAccount, Deposit, Withdraw** and **CloseAccount**. For each type of request messages, we present an example request message and the corresponding response message.

B.2.1 CreateNewAccount Operation

The **CreateNewAccount** operation is used to create a new account in banking service. An example of such a request and the corresponding response messages are following

CreateNewAccount request

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:createNewAccount
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <arg0>
                            <accountName>John Taaffe</accountName>
                            <accountNumber>229464651</accountNumber>
                            <accountType>SAVINGS</accountType>
                            <balance>415.66</balance>
                     </arg0>
              </ns2:createNewAccount>
       </S:Body>
</S:Envelope>
```

CreateNewAccount response

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:createNewAccountResponse
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <return>
                            <accountName>default</accountName>
                            <accountNumber>229464651</accountNumber>
                            <balance>0.0</balance>
                            <message>Sorry, the account already exists</message>
                            <result>AlreadyExists</result>
                     </return>
              </ns2:createNewAccountResponse>
       </S:Body>
</S:Envelope>
```

B.2.2 GetAccount Operation

The `GetAccount` operation is used to find the details of an existing account in banking service. An example of such a request and the corresponding response messages are following

GetAccount request

```
<S:Envelope
	xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
	xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
	<SOAP-ENV:Header/>
	<S:Body>
		<ns2:getAccount
			xmlns:ns2="http://publisher.webservice.bank.com/">
			<arg0>
				<accountNumber>1647664720</accountNumber>
			</arg0>
		</ns2:getAccount>
	</S:Body>
</S:Envelope>
```

GetAccount response

```
<S:Envelope
	xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
	xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
	<SOAP-ENV:Header/>
	<S:Body>
		<ns2:getAccountResponse
			xmlns:ns2="http://publisher.webservice.bank.com/">
			<return>
				<accountName>Judy Blume</accountName>
				<accountNumber>1647664720</accountNumber>
				<balance>99978.0</balance>
				<result>SUCCESS</result>
			</return>
		</ns2:getAccountResponse>
	</S:Body>
</S:Envelope>
```

B.2.3 Deposit Operation

The `Deposit` operation is used to deposit money to an existing account in banking service. An example of such a request and the corresponding response messages are following

Deposit request

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:deposit
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <arg0>
                            <accountNumber>1568941178</accountNumber>
                            <amount>6319</amount>
                            <transactionDate>2005-08-21</transactionDate>
                     </arg0>
              </ns2:deposit>
       </S:Body>
</S:Envelope>
```

Deposit response

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:depositResponse
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <return>
                            <accountNumber>1568941178</accountNumber>
                            <balance>16106.00</balance>
                            <message>The deposit request of amount 6319 has been
                            successful</message>
                            <result>SUCCESS</result>
                     </return>
              </ns2:depositResponse>
       </S:Body>
</S:Envelope>
```

B.2.4 Withdraw Operation

The `Withdraw` operation is used to withdraw money from an existing account if the account has sufficient amount of money in banking service. An example of such a request and the corresponding response messages are following

Withdraw request

```
<S:Envelope
      xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Header/>
      <S:Body>
            <ns2:withdraw
                  xmlns:ns2="http://publisher.webservice.bank.com/">
                  <arg0>
                        <accountNumber>229464651</accountNumber>
                        <amount>5411.00</amount>
                        <transactionDate>2010-11-24</transactionDate>
                  </arg0>
            </ns2:withdraw>
      </S:Body>
</S:Envelope>
```

Withdraw response

```
<S:Envelope
      xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Header/>
      <S:Body>
            <ns2:withdrawResponse
                  xmlns:ns2="http://publisher.webservice.bank.com/">
                  <return>
                        <accountNumber>229464651</accountNumber>
                        <balance>0.00</balance>
                        <message>The withdraw request of amount 5411 has been
                        successful</message>
                        <result>SUCCESS</result>
                  </return>
            </ns2:withdrawResponse>
      </S:Body>
</S:Envelope>
```

B.2.5 CloseAccount Operation

The `CloseAccount` operation is used to close an existing account in banking service. An example of such a request and the corresponding response messages are following

CloseAccount request

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:closeAccount
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <arg0>
                            <accountNumber>4114286369</accountNumber>
                            <transactionDate>2011-11-24</transactionDate>
                     </arg0>
              </ns2:closeAccount>
       </S:Body>
</S:Envelope>
```

CloseAccount response

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Header/>
       <S:Body>
              <ns2:closeAccountResponse
                     xmlns:ns2="http://publisher.webservice.bank.com/">
                     <return>
                            <accountNumber>4114286369</accountNumber>
                            <message>Sorry, no such account is available</message>
                            <result>NoSuchAccount</result>
                     </return>
              </ns2:closeAccountResponse>
       </S:Body>
</S:Envelope>
```

B.3 Twitter

The Twitter example trace contains six types of request messages, which are **Friendshipsshow, Statusesupdate, Searchtweets, StatusesShow**, **statusesuser_timeline-jsonuser_id** and **statusesuser_timeline_jsonscreen_name**. For each type of request messages, we present an example request message and the corresponding response message.

B.3.1 Friendshipsshow Operation

The **Friendshipsshow** operation is used to find the detailed information about the relationship between two arbitrary users. An example of such a request and the corresponding response messages are following

Friendshipsshow request

```
GET https://api.twitter.com/1.1/friendships/show.json?source_id=524642832&
target_id=133093395&include_entities=1&include_rts=1
```

Friendshipsshow response


```
{
  "relationship": {
    "source": {
      "id": 524642832,
      "blocking": null,
      "notifications_enabled": null,
      "following": false,
      "followed_by": false,
      "screen_name": "_abbiecornish",
      "marked_spam": null,
      "can_dm": false,
      "id_str": "524642832",
      "all_replies": null,
      "want_retweets": null
    },
    "target": {
      "id": 133093395,
      "following": false,
      "followed_by": false,
      "screen_name": "russellcrowe",
      "id_str": "133093395"
```

```
   }
  }
}
```

B.3.2 Statusesupdate Operation

The **Statusesupdate** operation is used to update the current status of an authenticated user (*i.e.*, Tweeting). An example of such a request and the corresponding response messages are following

Statusesupdate request

```
POST https://api.twitter.com/1.1/statuses/update.json status=circumaviator
\%20laughingly \%20unkemptness\%20scleretinite\%20nonthinking\%20catalogicl
\%20vicissitudinous\%20bundook\%20sagenite\%20feltyfare.&include_entities=1&
include_rts=1
```

Statusesupdate response

```
{
  "contributors": null,
  "text": "circumaviator laughingly unkemptness scleretinite nonthinking
  catalogical vicissitudinous bundook sagenite feltyfare.",
  "geo": null,
  "retweeted": false,
  "in_reply_to_screen_name": null,
  "truncated": false,
  "lang": "en",
  "entities": {
    "symbols": [],
    "urls": [],
    "hashtags": [],
    "user_mentions": []
  },
  "in_reply_to_status_id_str": null,
  "id": 425241003532894200,
  "source": "<a href=\"http://www.baidu.com\" rel=\"nofollow\">traffic_gen</a>",
  "in_reply_to_user_id_str": null,
  "favorited": false,
  "in_reply_to_status_id": null,
  "retweet_count": 0,
```

```
"created_at": "Mon Jan 20 12:18:50 +0000 2014",
"in_reply_to_user_id": null,
"favorite_count": 0,
"id_str": "425241003532894208",
"place": null,
"user": {
 "location": "",
 "default_profile": true,
 "profile_background_tile": false,
 "statuses_count": 179,
 "lang": "en-gb",
 "profile_link_color": "0084B4",
 "id": 2300564737,
 "following": false,
 "protected": false,
 "favourites_count": 0,
 "profile_text_color": "333333",
 "description": "",
 "verified": false,
 "contributors_enabled": false,
 "profile_sidebar_border_color": "C0DEED",
 "name": "miao",
 "profile_background_color": "C0DEED",
 "created_at": "Mon Jan 20 01:18:51 +0000 2014",
 "default_profile_image": true,
 "followers_count": 0,
 "profile_image_url_https": "https://abs.twimg.com/sticky/default_profile_
 images/default_profile_2_normal.png",
 "geo_enabled": false,
 "profile_background_image_url": "http://abs.twimg.com/images/themes/theme1/
 bg.png",
 "profile_background_image_url_https": "https://abs.twimg.com/images/themes/
 theme1/bg.png",
 "follow_request_sent": false,
 "entities": {
  "description": {
   "urls": []
  }
 },
 "url": null,
 "utc_offset": null,
 "time_zone": null,
 "notifications": false,
 "profile_use_background_image": true,
 "friends_count": 2,
```

```
    "profile_sidebar_fill_color": "DDEEF6",
    "screen_name": "roundhole19",
    "id_str": "2300564737",
    "profile_image_url": "http://abs.twimg.com/sticky/default_profile_images/
    default_profile_2_normal.png",
    "listed_count": 0,
    "is_translator": false
  },
  "coordinates": null
}
```


B.3.3 Searchtweets Operation

The **Searchtweets** operation is used to search with Tweets since the first one posted in 2006. An example of such a request and the corresponding response messages are following

Searchtweets request

```
GET https://api.twitter.com/1.1/search/tweets.json?q=
Majestic\%20http\%3A\%2F\%2Ft.co\%2FAzVLm6NhFa&with_twitter_user_id=
true&include_entities=1&include_rts=1
```

Searchtweets response


```
{
  "statuses": [
    {
      "retweeted": false,
      "in_reply_to_screen_name": null,
      "possibly_sensitive": false,
      "truncated": false,
      "lang": "pl",
      "in_reply_to_status_id_str": null,
      "id": 422981825489682400,
      "in_reply_to_user_id_str": null,
      "in_reply_to_status_id": null,
      "created_at": "Tue Jan 14 06:41:40 +0000 2014",
      "favorite_count": 0,
      "place": null,
      "coordinates": null,
      "metadata": {
        "result_type": "recent",
```

```
  "iso_language_code": "pl"
},
"text": "RT @Ryan_Kwanten: Majestic... http://t.co/AzVLm6NhFa",
"contributors": null,
"retweeted_status": {
  "contributors": null,
  "text": "Majestic... http://t.co/AzVLm6NhFa",
  "geo": null,
  "retweeted": false,
  "in_reply_to_screen_name": null,
  "possibly_sensitive": false,
  "truncated": false,
  "lang": "pl",
  "entities": {
    "symbols": [],
    "urls": [],
    "hashtags": [],
    "media": [
     {
       "sizes": {
         "small": {
           "w": 340,
           "resize": "fit",
           "h": 453
         },
         "thumb": {
           "w": 150,
           "resize": "crop",
           "h": 150
         },
         "medium": {
           "w": 600,
           "resize": "fit",
           "h": 800
         },
         "large": {
           "w": 1024,
           "resize": "fit",
           "h": 1365
         }
       },
       "id": 408740134721310700,
       "media_url_https": "https://pbs.twimg.com/media/Bawi_KeCUAAaTMa.jpg",
       "media_url": "http://pbs.twimg.com/media/Bawi_KeCUAAaTMa.jpg",
       "expanded_url": "http://twitter.com/Ryan_Kwanten/status/
```

```
      408740134712922112/photo/1",
      "indices": [
       12,
       34
      ],
      "id_str": "408740134721310720",
      "type": "photo",
      "display_url": "pic.twitter.com/AzVLm6NhFa",
      "url": "http://t.co/AzVLm6NhFa"
    }
  ],
  "user_mentions": []
},
"in_reply_to_status_id_str": null,
"id": 408740134712922100,
"source": "web",
"in_reply_to_user_id_str": null,
"favorited": false,
"in_reply_to_status_id": null,
"retweet_count": 57,
"created_at": "Thu Dec 05 23:30:17 +0000 2013",
"in_reply_to_user_id": null,
"favorite_count": 209,
"id_str": "408740134712922112",
"place": null,
"user": {
  "location": "",
  "default_profile": false,
  "profile_background_tile": true,
  "statuses_count": 354,
  "lang": "en",
  "profile_link_color": "0084B4",
  "profile_banner_url": "https://pbs.twimg.com/profile_banners
  /158900583/1364356693",
  "id": 158900583,
  "following": false,
  "protected": false,
  "favourites_count": 91,
  "profile_text_color": "333333",
  "description": "The official Ryan Kwanten Twitter account.",
  "verified": true,
  "contributors_enabled": false,
  "profile_sidebar_border_color": "FFFFFF",
  "name": "Ryan Kwanten",
  "profile_background_color": "C0DEED",
```

```
  "created_at": "Wed Jun 23 23:35:04 +0000 2010",
  "default_profile_image": false,
  "followers_count": 178699,
  "profile_image_url_https": "https://pbs.twimg.com/profile_images
  /3436412603/2341fc2d7345b1e9b74f5ab72300ddb6_normal.jpeg",
  "geo_enabled": false,
  "profile_background_image_url": "http://a0.twimg.com/profile_background_
  images/825737319/c30b658b3253133e9fa0c9a7000f2fff.jpeg",
  "profile_background_image_url_https": "https://si0.twimg.com/
  profile_background_images/825737319/c30b658b3253133e9fa0c9a7000f2fff.jpeg",
  "follow_request_sent": false,
  "entities": {
   "description": {
    "urls": []
   },
   "url": {
    "urls": [
     {
      "expanded_url": "https://www.facebook.com/RyanKwanten",
      "indices": [
       0,
       23
      ],
      "display_url": "facebook.com/RyanKwanten",
      "url": "https://t.co/HxItxdgm89"
     }
    ]
   }
  },
  "url": "https://t.co/HxItxdgm89",
  "utc_offset": null,
  "time_zone": null,
  "notifications": false,
  "profile_use_background_image": true,
  "friends_count": 14,
  "profile_sidebar_fill_color": "DDEEF6",
  "screen_name": "Ryan_Kwanten",
  "id_str": "158900583",
  "profile_image_url": "http://pbs.twimg.com/profile_images/
  3436412603/2341fc2d7345b1e9b74f5ab72300ddb6_normal.jpeg",
  "listed_count": 2063,
  "is_translator": false
},
"coordinates": null,
"metadata": {
```

```
    "result_type": "recent",
    "iso_language_code": "pl"
   }
},
"geo": null,
"entities": {
  "symbols": [],
  "urls": [],
  "hashtags": [],
  "media": [
    {
      "sizes": {
        "small": {
          "w": 340,
          "resize": "fit",
          "h": 453
        },
        "thumb": {
          "w": 150,
          "resize": "crop",
          "h": 150
        },
        "medium": {
          "w": 600,
          "resize": "fit",
          "h": 800
        },
        "large": {
          "w": 1024,
          "resize": "fit",
          "h": 1365
        }
      },
      "id": 408740134721310700,
      "media_url_https": "https://pbs.twimg.com/media/Bawi_KeCUAAaTMa.jpg",
      "media_url": "http://pbs.twimg.com/media/Bawi_KeCUAAaTMa.jpg",
      "expanded_url": "http://twitter.com/Ryan_Kwanten/status
      /408740134712922112/photo/1",
      "indices": [
        30,
        52
      ],
      "id_str": "408740134721310720",
      "type": "photo",
      "display_url": "pic.twitter.com/AzVLm6NhFa",
```

```
      "url": "http://t.co/AzVLm6NhFa"
    }
   ],
   "user_mentions": [
    {
      "id": 158900583,
      "name": "Ryan Kwanten",
      "indices": [
       3,
       16
      ],
      "screen_name": "Ryan_Kwanten",
      "id_str": "158900583"
    }
   ]
  },
  "source": "<a href=\"https://mobile.twitter.com\"
  rel=\"nofollow\">Mobile Web (M5)</a>",
  "favorited": false,
  "in_reply_to_user_id": null,
  "retweet_count": 57,
  "id_str": "422981825489682432",
  "user": {
   "location": "",
   "default_profile": false,
   "profile_background_tile": true,
   "statuses_count": 9594,
   "lang": "en",
   "profile_link_color": "510969",
   "profile_banner_url": "https://pbs.twimg.com/profile_banners/310652200/
   1389892120",
   "id": 310652200,
   "following": false,
   "protected": false,
   "favourites_count": 6026,
   "profile_text_color": "6814A8",
   "description": "I am. Whatever you say I am, if I wasn't. Then
   why would I say I am.",
   "verified": false,
   "contributors_enabled": false,
   "profile_sidebar_border_color": "000000",
   "name": " Hayley",
   "profile_background_color": "FFFFFF",
   "created_at": "Sat Jun 04 03:39:01 +0000 2011",
   "default_profile_image": false,
```

```
      "followers_count": 1331,
      "profile_image_url_https": "https://pbs.twimg.com/profile_images/
      378800000831774897/0e565bd296ea03a217e9abbd2c2bee12_normal.jpeg",
      "geo_enabled": true,
      "profile_background_image_url": "http://a0.twimg.com/profile_background
      _images/378800000071035705/1c9368d724131c7136888ab585104c38.jpeg",
      "profile_background_image_url_https": "https://si0.twimg.com/profile_
      background_images/378800000071035705/1c9368d724131c7136888ab585104c38
      .jpeg",
      "follow_request_sent": false,
      "entities": {
        "description": {
          "urls": []
        }
      },
      "url": null,
      "utc_offset": null,
      "time_zone": null,
      "notifications": false,
      "profile_use_background_image": true,
      "friends_count": 1617,
      "profile_sidebar_fill_color": "9A14C7",
      "screen_name": "hkeevers",
      "id_str": "310652200",
      "profile_image_url": "http://pbs.twimg.com/profile_images/378800000831774897/
      0e565bd296ea03a217e9abbd2c2bee12_normal.jpeg",
      "listed_count": 2,
      "is_translator": false
    }
  }
],
"search_metadata": {
  "since_id": 0,
  "count": 15,
  "max_id": 422981825489682400,
  "refresh_url": "?since_id=422981825489682432&q=Majestic\%20http\%3A\%2F\%2Ft.co
  \%2FAzVLm6NhFa&include_entities=1",
  "query": "Majestichttp\%3A\%2F\%2Ft.co\%2FAzVLm6NhFa",
  "max_id_str": "422981825489682432",
  "since_id_str": "0",
  "completed_in": 0.041
}
}
```

B.3.4 StatusesShow Operation

The `StatusesShow` operation is used to get a single Tweet of an account (specified by id). An example of such a request and the corresponding response messages are following

StatusesShow request

```
GET https://api.twitter.com/1.1/statuses/show/423234210736787456.json?
include_entities=1&include_rts=1
```

StatusesShow response


```
{
  "contributors": null,
  "text": "@Headshambles Don't drop your guitar!",
  "geo": null,
  "retweeted": false,
  "in_reply_to_screen_name": "Headshambles",
  "truncated": false,
  "lang": "en",
  "entities": {
    "symbols": [],
    "urls": [],
    "hashtags": [],
    "user_mentions": [
      {
        "id": 710955037,
        "name": "Mark Hamilton",
        "indices": [
          0,
          13
        ],
        "screen_name": "Headshambles",
        "id_str": "710955037"
      }
    ]
  },
  "in_reply_to_status_id_str": "423232313636311040",
  "id": 423234210736787460,
  "source": "<a href=\"https://about.twitter.com/products/tweetdeck\"
  rel=\"nofollow\">TweetDeck</a>",
  "in_reply_to_user_id_str": "710955037",
```

```
"favorited": false,
"in_reply_to_status_id": 423232313636311040,
"retweet_count": 0,
"created_at": "Tue Jan 14 23:24:33 +0000 2014",
"in_reply_to_user_id": 710955037,
"favorite_count": 0,
"id_str": "423234210736787456",
"place": null,
"user": {
 "location": "Sydney",
 "default_profile": false,
 "profile_background_tile": false,
 "statuses_count": 11317,
 "lang": "en",
 "profile_link_color": "990000",
 "profile_banner_url": "https://pbs.twimg.com/profile_banners/22711711/1356051799",
 "id": 22711711,
 "following": false,
 "protected": false,
 "favourites_count": 1941,
 "profile_text_color": "333333",
 "description": "January 9-26 in Sydney. Music, theatre, opera, dance, visual arts,
 ideas, free & family events. It's big. This is our city in summer. #sydfest",
 "verified": true,
 "contributors_enabled": false,
 "profile_sidebar_border_color": "FFFFFF",
 "name": "Sydney Festival",
 "profile_background_color": "86CEDE",
 "created_at": "Wed Mar 04 01:14:38 +0000 2009",
 "default_profile_image": false,
 "followers_count": 47853,
 "profile_image_url_https": "https://pbs.twimg.com/profile_images/2505506864/
 tekyg38071cjub5w1n90_normal.png",
 "geo_enabled": true,
 "profile_background_image_url": "http://a0.twimg.com/profile_background_
 images/378800000103520193/ef9daab733b72f6056e3404f8941dfdb.jpeg",
 "profile_background_image_url_https": "https://si0.twimg.com/profile_
 background_images/378800000103520193/
 ef9daab733b72f6056e3404f8941dfdb.jpeg",
 "follow_request_sent": false,
 "entities": {
  "description": {
   "urls": []
  },
  "url": {
```

```
      "urls": [
       {
         "expanded_url": "http://www.sydneyfestival.org.au",
         "indices": [
           0,
           22
         ],
         "display_url": "sydneyfestival.org.au",
         "url": "http://t.co/1A76sKRwUL"
       }
      ]
     }
   },
   "url": "http://t.co/1A76sKRwUL",
   "utc_offset": 39600,
   "time_zone": "Sydney",
   "notifications": false,
   "profile_use_background_image": true,
   "friends_count": 6321,
   "profile_sidebar_fill_color": "FAD105",
   "screen_name": "sydney_festival",
   "id_str": "22711711",
   "profile_image_url": "http://pbs.twimg.com/profile_images/2505506864/
   tekyg38071cjub5w1n90_normal.png",
   "listed_count": 1088,
   "is_translator": false
  },
  "coordinates": null
}
```


B.3.5 Statusesuser_timelinejsonuser_id Operation

The `Statusesuser_timelinejsonuser_id` operation is used to get a collection of the most recent Tweets posted by the user indicated by the *id*. An example of such a request and the corresponding response messages are following

Statusesuser_timelinejsonuser_id request

```
GET https://api.twitter.com/1.1/statuses/user_timeline.json?user_id=27
042513&include_my_retweet=true&include_entities=1&include_rts=1
```

Statusesuser_timelinejsonuser_id response


```
[
{
   "contributors": null,
   "text": "Starry night! http://t.co/kGhhHRcBVi",
   "geo": null,
   "retweeted": false,
   "in_reply_to_screen_name": null,
   "possibly_sensitive": false,
   "truncated": false,
   "lang": "en",
   "entities": {
     "symbols": [],
     "urls": [
       {
         "expanded_url": "http://instagram.com/p/jXcrnwihLO/",
         "indices": [
           14,
           36
         ],
         "display_url": "instagram.com/p/jXcrnwihLO/",
         "url": "http://t.co/kGhhHRcBVi"
       }
     ],
     "hashtags": [],
     "user_mentions": []
   },
   "in_reply_to_status_id_str": null,
   "id": 425024613622771700,
   "source": "<a href=\"http://instagram.com\" rel=\"nofollow\">Instagram</a>",
   "in_reply_to_user_id_str": null,
   "favorited": false,
   "in_reply_to_status_id": null,
   "retweet_count": 152,
   "created_at": "Sun Jan 19 21:58:59 +0000 2014",
   "in_reply_to_user_id": null,
   "favorite_count": 438,
   "id_str": "425024613622771713",
   "place": null,
   "user": {
     "location": "Sydney ",
     "default_profile": false,
     "profile_background_tile": false,
     "statuses_count": 751,
```

```
    "lang": "en",
    "profile_link_color": "2FC2EF",
    "id": 27042513,
    "following": false,
    "protected": false,
    "favourites_count": 1,
    "profile_text_color": "666666",
    "description": "Instagram: @TheHughJackman",
    "verified": true,
    "contributors_enabled": false,
    "profile_sidebar_border_color": "181A1E",
    "name": "Hugh Jackman",
    "profile_background_color": "1A1B1F",
    "created_at": "Fri Mar 27 16:45:10 +0000 2009",
    "default_profile_image": false,
    "followers_count": 3614701,
    "profile_image_url_https": "https://pbs.twimg.com/profile_images/
    124111564/08_Jackman_085_normal.jpg",
    "geo_enabled": false,
    "profile_background_image_url": "http://abs.twimg.com/images/themes/
    theme9/bg.gif",
    "profile_background_image_url_https": "https://abs.twimg.com/images/themes/
    theme9/bg.gif",
    "follow_request_sent": false,
    "entities": {
     "description": {
      "urls": []
     }
    },
    "url": null,
    "utc_offset": -28800,
    "time_zone": "Pacific Time (US & Canada)",
    "notifications": false,
    "profile_use_background_image": true,
    "friends_count": 21,
    "profile_sidebar_fill_color": "252429",
    "screen_name": "RealHughJackman",
    "id_str": "27042513",
    "profile_image_url": "http://pbs.twimg.com/profile_images/124111564/
    08_Jackman_085_normal.jpg",
    "listed_count": 22199,
    "is_translator": false
   },
   "coordinates": null
 },
```

```
  ...................
  ...................
  ]
```

B.3.6 Statusesuser_timeline_jsonscreen_name Operation

The **Statusesuser_timeline_jsonscreen_name** operation is used to get a collection of the most recent Tweets posted by the user indicated by the *screen_name*. An example of such a request and the corresponding response messages are following

Statusesuser_timeline_jsonscreen_name request

```
GET https://api.twitter.com/1.1/statuses/user_timeline.json?screen_
name=RafaelNadal&include_my_retweet=true&include_entities=1&include_rts=1
```

Statusesuser_timeline_jsonscreen_name response


```
[
{
   "contributors": null,
   "text": "Recuperando fuerzas después de entrenar bajo el calor
   australiano!! \nRecharging batteries after practice with the...
   http://t.co/morVrqpcVa",
   "geo": null,
   "retweeted": false,
   "in_reply_to_screen_name": null,
   "possibly_sensitive": false,
   "truncated": false,
   "lang": "es",
   "entities": {
     "symbols": [],
     "urls": [
       {
         "expanded_url": "http://fb.me/VeDTBE1T",
         "indices": [
           117,
           139
         ],
         "display_url": "fb.me/VeDTBE1T",
         "url": "http://t.co/morVrqpcVa"
```

```
    }
   ],
   "hashtags": [],
   "user_mentions": []
  },
  "in_reply_to_status_id_str": null,
  "id": 423367086144892900,
  "source": "<a href=\"http://www.facebook.com/twitter\" rel=\"nofollow\">
  Facebook</a>",
  "in_reply_to_user_id_str": null,
  "favorited": false,
  "in_reply_to_status_id": null,
  "retweet_count": 364,
  "created_at": "Wed Jan 15 08:12:33 +0000 2014",
  "in_reply_to_user_id": null,
  "favorite_count": 502,
  "id_str": "423367086144892929",
  "place": null,
  "user": {
   "location": "Manacor",
   "default_profile": false,
   "profile_background_tile": false,
   "statuses_count": 1437,
   "lang": "es",
   "profile_link_color": "050505",
   "id": 344634424,
   "following": false,
   "protected": false,
   "favourites_count": 16,
   "profile_text_color": "0A0A0A",
   "description": "Tennis player.",
   "verified": true,
   "contributors_enabled": false,
   "profile_sidebar_border_color": "FFFFFF",
   "name": "Rafa Nadal",
   "profile_background_color": "C0DEED",
   "created_at": "Fri Jul 29 10:44:02 +0000 2011",
   "default_profile_image": false,
   "followers_count": 5625513,
   "profile_image_url_https": "https://pbs.twimg.com/profile_images/
   1475846482/Imagen_perfil_normal.jpg",
   "geo_enabled": false,
   "profile_background_image_url": "http://a0.twimg.com/profile_background_
   images/378800000115991559/9bad21643487dc1d2b2b017458b25131.jpeg",
   "profile_background_image_url_https": "https://si0.twimg.com/profile_
```

```
    background_images/378800000115991559/9bad21643487dc1d2b2b017458b25131.jpeg",
    "follow_request_sent": false,
    "entities": {
      "description": {
        "urls": []
      },
      "url": {
        "urls": [
          {
            "expanded_url": "http://www.rafaelnadal.com/",
            "indices": [
              0,
              22
            ],
            "display_url": "rafaelnadal.com",
            "url": "http://t.co/ZU7sGR5Dct"
          }
        ]
      }
    },
    "url": "http://t.co/ZU7sGR5Dct",
    "utc_offset": -36000,
    "time_zone": "Hawaii",
    "notifications": false,
    "profile_use_background_image": true,
    "friends_count": 74,
    "profile_sidebar_fill_color": "7AC0D6",
    "screen_name": "RafaelNadal",
    "id_str": "344634424",
    "profile_image_url": "http://pbs.twimg.com/profile_images/
    1475846482/Imagen_perfil_normal.jpg",
    "listed_count": 18297,
    "is_translator": false
  },
  "coordinates": null
},

.....................
.....................
]
```

B.4 GoogleBooks

The GoogleBooks example trace contains two types of request messages, which are `Search_Volume` and `Search_Bookshelf`. For each type of request messages, we present an example request message and the corresponding response message.

B.4.1 Search_Volume Operation

The `Search_Volume` operation is used to search books in the GoogleBooks based on the keywords or retrieve information about a specific volume using *volume id*. An example of such a request and the corresponding response messages are following

GoogleBooks Search_Volume Request

```
https://www.googleapis.com/books/v1/volumes?q=Tragedy\%20in\%20the
\%20Victorian\%20Novel&maxResults=40
```

GoogleBooks Search_Volume Response


```
{
  "kind": "books#volumes",
  "totalItems": 6454,
  "items": [
   {
     "kind": "books#volume",
     "id": "SQA4AAAAIAAJ",
     "etag": "+h0RNCgaeW8",
     "selfLink": "https://www.googleapis.com/books/v1/volumes/SQA4AAAAIAAJ",
     "volumeInfo": {
       "title": "Tragedy in the Victorian Novel",
       "subtitle": "Theory and Practice in the Novels of George Eliot, Thomas Hardy
       and Henry James",
       "authors": [
         "Jeannette King"
       ],
       "publisher": "CUP Archive",
       "publishedDate": "1979",
       "description": "Dr King examines the rise of the novel in the nineteenth
       century, and how it came to embody the tragic vision of life which had
       previously been the domain of drama. Dr King focuses on three novelists,
       George Eliot, Thomas Hardy and Henry James.",
```

```
  "industryIdentifiers": [
   {
    "type": "ISBN_10",
    "identifier": "0521297443"
   },
   {
    "type": "ISBN_13",
    "identifier": "9780521297448"
   }
  ],
  "readingModes": {
   "text": false,
   "image": true
  },
  "pageCount": 192,
  "printType": "BOOK",
  "categories": [
   "Literary Criticism"
  ],
  "maturityRating": "NOT_MATURE",
  "allowAnonLogging": false,
  "contentVersion": "0.0.1.0.preview.1",
  "imageLinks": {
   "smallThumbnail": "http://books.google.com/books/content?id=
   SQA4AAAAIAAJ&printsec=frontcover&img=1&zoom=5&edge=curl&source=gbs_api",
   "thumbnail": "http://books.google.com/books/content?id=
   SQA4AAAAIAAJ&
   printsec=frontcover&img=1&zoom=1&edge=curl&source=gbs_api"
  },
  "language": "en",
  "previewLink": "http://books.google.com.au/books?id=SQA4AAAAIAAJ
  &printsec=frontcover&dq=Tragedy+in+the+Victorian+Novel&
  hl=&cd=1&source=gbs_api",
  "infoLink": "http://books.google.com.au/books?id=SQA4AAAAIAAJ&dq=
  Tragedy+in+the+Victorian+Novel&hl=&source=gbs_api",
  "canonicalVolumeLink": "https://books.google.com/books/about/Tragedy_
  in_the_Victorian_Novel.html?hl=&id=SQA4AAAAIAAJ"
 },
 "saleInfo": {
  "country": "AU",
  "saleability": "NOT_FOR_SALE",
  "isEbook": false
 },
 "accessInfo": {
  "country": "AU",
```

```
      "viewability": "PARTIAL",
      "embeddable": true,
      "publicDomain": false,
      "textToSpeechPermission": "ALLOWED",
      "epub": {
        "isAvailable": false
      },
      "pdf": {
        "isAvailable": false
      },
      "webReaderLink": "http://play.google.com/books/reader?id=SQA4AAAAIAAJ&hl=
      &printsec=frontcover&source=gbs_api",
      "accessViewStatus": "SAMPLE",
      "quoteSharingAllowed": false
    },
    "searchInfo": {
      "textSnippet": "Dr King examines the rise of the novel in the nineteenth
      century, and how it came to embody the tragic vision of life which had
      previously been the domain of drama."
    }
   }
   ..................................
   ..................................
 ]
}
```


B.4.2 Search_Bookshelf Operation

The `Search_Bookshelf` operation is used to get a collection of a user's public bookshelves for the user indicated by the *user ID*. An example of such a request and the corresponding response messages are following

GoogleBooks Search_Bookshelf Request


```
https://www.googleapis.com/books/v1/users/107782646712117400162/
bookshelves/0?key=**************
```


Search_Bookshelf Response


```
{
 "kind": "books#bookshelf",
 "id": 0,
 "selfLink": "https://www.googleapis.com/books/v1/users/107782646712117400162/
```

```
  bookshelves/0",
  "title": "Favorites",
  "access": "PUBLIC",
  "updated": "2019-10-01T13:29:39.162Z",
  "created": "2019-10-01T13:29:39.162Z",
  "volumeCount": 7,
  "volumesLastUpdated": "2019-10-01T13:29:39.160Z"
}
```

Bibliography

[1] CA Technologies Inc. CA Identity Manager. [Accessed: Jan. 12, 2019]. [Online]. Available: https://www.ca.com/au/products/ca-identity-manager.html

[2] M. Du, J.-G. Schneider, C. Hine, J. Grundy, and S. Versteeg, "Generating service models by trace subsequence substitution," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. Vancouver, British Columbia, Canada: ACM, Jun. 2013, pp. 123–132.

[3] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Enterprise software service emulation: constructing large-scale testbeds," in *Proceedings of the IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*. Austin, Texas, USA: IEEE, May 2016, pp. 56–62.

[4] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Proceedings of the International Conference on Object-oriented Programming Systems Languages and Applications*. Vancouver, British Columbia, Canada: ACM, Oct. 2004, pp. 236–246.

[5] T. Mackinnon, S. Freeman, and P. Craig, *Endo-testing: Unit Testing with Mock Objects*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[6] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. Boston, Massachusetts, USA: USENIX Association, Jun. 2001, pp. 1–14.

[7] P. Li, "Selecting and using virtualization solutions: our experiences with vmware and virtualbox," *Journal of Computing Sciences in Colleges*, vol. 25, no. 3, pp. 11–17, 2010.

[8] J. Sanchez, "Squeezing virtual machines out [of] cpu cores," *VM Install*, Jun. 2011.

[9] M. Du, S. Versteeg, J.-G. Schneider, J. Han, and J. Grundy, "Interaction traces mining for efficient system responses generation," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–8, 2015.

[10] S. Versteeg, M. Du, J. Bird, J.-G. Schneider, J. Grundy, and J. Han, "Enhanced playback of automated service emulation models using entropy analysis," in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery (CSED)*. Austin, Texas, USA: IEEE, May 2016, pp. 49–55.

[11] S. Versteeg, M. Du, J.-G. Schneider, J. Grundy, J. Han, and M. Goyal, "Opaque service virtualisation: a practical tool for emulating endpoint systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*. Austin, Texas, USA: ACM, May 2016, pp. 202–211.

[12] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-oriented Architecture Best Practices*. Upper Saddle River, New Jersey, USA: Prentice-Hall, 2005.

[13] ABC News. Computer switch to blame for tunnel closures. [Online]. Available: https://www.abc.net.au/news/2012-10-30/computer-switch-to-blame-for-tunnel-closures/4342624

[14] Y.-Y. Su and J. Flinn, "Slingshot: Deploying stateful services in wireless hotspots," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. Seattle, Washington, USA: ACM, Jun. 2005, pp. 79–92.

[15] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[16] J. Gao, H.-S. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-based Software*. Norwood, Massachusetts, USA: Artech House, 2003.

[17] G. O'Regan, *Introduction to Software Quality*. Springer, 2014.

[18] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Old Tappan, New Jersey, USA: Addison-Wesley, 2015.

[19] J. Michelsen and J. English, *Service Virtualization: Reality Is Overrated*, 1st ed. Berkely, California, USA: Apress, 2012.

[20] W. Yeong, T. Howes, and S. Kille, "Lightweight directory access protocol," Internet Engineering Task Force (IETF), ISOC, Fremont, California, USA, RFC 1777, Mar. 1995. [Online]. Available: http://www.rfc-editor.org/info/rfc1777

[21] P. B. Gibbons, "A stub generator for multilanguage rpc in heterogeneous environments," *IEEE Transactions on Software Engineering*, no. 1, pp. 77–87, Jan. 1987.

[22] T. Freese, "Easymock: dynamic mock objects for junit," in *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2002)*. Alghero, Italy: Citeseer, May 2002, pp. 2–5.

[23] V. Massol and T. Husted, *JUnit in Action*. Greenwich, Connecticut, USA: Manning Publications Co., 2003.

[24] K. Lawton *et al.* (2001) Plex86. [Accessed: Jun. 14, 2019]. [Online]. Available: http://plex86.sourceforge.net/

[25] Docker. [Accessed: Jan. 15, 2019]. [Online]. Available: https://www.docker.com/

[26] OpenVz. [Accessed: Jan. 16, 2019]. [Online]. Available: https://openvz.org/

[27] Google lmctfy. [Accessed: Jan. 16, 2019]. [Online]. Available: https://github.com/google/lmctfy

[28] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Scalable emulation of enterprise systems," in *Proceedings of the Australian Software Engineering Conference*. Gold Coast, QLD, Australia: IEEE, Apr. 2009, pp. 142–151.

[29] CA Technologies Inc. CA Service Virtualization. [Accessed: Feb. 20, 2019]. [Online]. Available: https://www.ca.com/au/products/ca-service-virtualization.html

[30] ServiceV Pro. [Accessed: Feb. 20, 2019]. [Online]. Available: https://smartbear.com/product/ready-api/servicev/overview/

[31] IBM Service Virtualization. [Accessed: Feb. 20, 2019]. [Online]. Available: https://www.automation-consultants.com/products/ibm-products/ibm-service/

[32] Wiremock. [Accessed: Jun. 12, 2019]. [Online]. Available: https://github.com/tomakehurst/wiremock

[33] Hoverfly. [Accessed: Jun. 12, 2019]. [Online]. Available: https://github.com/SpectoLabs/hoverfly

[34] Citrus. [Accessed: Jun. 12, 2019]. [Online]. Available: https://citrusframework.org/docs/download/

[35] SoapUI. [Accessed: Feb. 20, 2019]. [Online]. Available: https://www.soapui.org/downloads/soapui.html

[36] Wilma. [Accessed: Jun. 12, 2019]. [Online]. Available: https://github.com/epam/Wilma

[37] H. F. Enişer and A. Sen, "Testing service oriented architectures using stateful service visualization via machine learning," in *Proceedings of the 13th International Workshop on Automation of Software Test*. Gothenburg, Sweden: ACM, May 2018, pp. 9–15.

[38] S. Ghosh, A. P. Mathur *et al.*, "Issues in testing distributed component-based systems," in *Proceedings of the 1st International ICSE Workshop on Testing Distributed Component-Based Systems*. Los Angeles, California, USA: IEEE, May 1999.

[39] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, Mar. 2016.

[40] A. Tridgell, "How samba was written," Aug. 2003. [Online]. Available: https://www.samba.org/ftp/tridge/misc/french_cafe.txt

[41] M. Mintz and A. Sayers, "MSN Messenger protocol," 2003. [Online]. Available: http://www.hypothetic.org/docs/msn/index.php

[42] "libyahoo2: A C library for Yahoo! Messenger," Jul. 2010. [Online]. Available: http://libyahoo2.sourceforge.net/

[43] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security.* Alexandria, Virginia, USA: ACM, Oct. 2007, pp. 317–329.

[44] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the 30th IEEE Symposium on Security and Privacy.* Berkeley, California, USA: IEEE, May 2009, pp. 110–125.

[45] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of 13th USENIX Security Symposium.* San Diego, California, USA: USENIX Association, Aug. 2004, pp. 321–336.

[46] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles.* Brighton, United Kingdom: ACM, Oct. 2005, pp. 133–147.

[47] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* Portland, Oregon, USA: IEEE Computer Society, Dec. 2004, pp. 221–232.

[48] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, "Automatic network protocol analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS).* San Diego, California, USA: The Internet Society, Feb. 2008, pp. 1–14.

[49] J. Lim, T. Reps, and B. Liblit, "Extracting output formats from executables," in *Proceedings of the 13th Working Conference on Reverse Engineering.* Benevento, Italy: IEEE, Oct. 2006, pp. 167–178.

[50] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS).* San Diego, California, USA: The Internet Society, Feb. 2008, pp. 1–15.

[51] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Proceedings of the 14th Eu-*

ropean Symposium on Research in Computer Security. Saint-Malo, France: Springer, Sep. 2009, pp. 200–215.

[52] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th USENIX Security Symposium*. San Diego, California, USA: USENIX Association, Aug. 2004, pp. 321–336.

[53] McAfee LLC. (2004) Network Protocol Analysis using Bioinformatics Algorithms. Santa Clara, California, USA. [Accessed: Aug. 20, 2019]. [Online]. Available: www.4tphi.net/~awalters/PI/pi.pdf

[54] M. Nei, F. Tajima, and Y. Tateno, "Accuracy of estimated phylogenetic trees from molecular data," *Journal of Molecular Evolution*, vol. 19, no. 2, pp. 153–170, Mar. 1983.

[55] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium*. Boston, Massachusetts, USA: USENIX Association, Aug. 2007, pp. 1–14.

[56] T. Krueger, N. Krämer, and K. Rieck, "Asap: Automatic semantics-aware analysis of network payloads," in *Proceedings of the International Workshop on Privacy and Security Issues in Data Mining and Machine Learning*. Barcelona, Spain: Springer, Sep. 2010, pp. 50–63.

[57] J. Duchene, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 53–68, Feb. 2018.

[58] J. Antunes, N. Neves, and P. Verissimo, "Reverse engineering of protocols from network traces," in *Proceedings of the 18th Working Conference on Reverse Engineering*. Limerick, Ireland: IEEE, Oct. 2011, pp. 169–178.

[59] Y. Wang, N. Zhang, Y.-M. Wu, and B.-B. Su, "Protocol specification inference based on keywords identification," in *Advanced Data Mining and Applications (ADMA). LNCS*, H. Motoda, Z. Wu, L. Cao, O. Zaiane, M. Yao, and W. Wang, Eds. Berlin, Heidelberg: Springer, Dec. 2013, vol. 8347, pp. 443–454.

[60] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, no. 5323, pp. 34–35, Nov. 1971.

[61] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata Studies*, vol. 34, pp. 129–153, 1956.

[62] M. Määttä and T. Räty, "A modelling approach for monitoring sequence activities in diverse environments," in *Proceedings of the 9th International Conference on Digital Information Management (ICDIM).* Phitsanulok, Thailand: IEEE, Sep. 2014, pp. 33–38.

[63] A. Tongaonkar, R. Keralapura, and A. Nucci, "Santaclass: A self adaptive network traffic classification system," in *Proceedings of the International Federation for Information Processing (IFIP) Networking Conference.* Brooklyn, New York, USA: IEEE, May 2013, pp. 1–9.

[64] A. Tongaonkar, R. Torres, M. Iliofotou, R. Keralapura, and A. Nucci, "Towards self adaptive network traffic classification," *Computer Communications*, vol. 56, pp. 35–46, Feb. 2015.

[65] B.-C. Li and S.-Z. Yu, "Keyword mining for private protocols tunneled over websocket," *IEEE Communications Letters*, vol. 20, no. 7, pp. 1337–1340, May 2016.

[66] I. Fette and A. Melnikov, "The websocket protocol," Internet Engineering Task Force (IETF), ISOC, Fremont, California, USA, RFC 6455, Dec. 2011. [Online]. Available: https://www.ietf.org/rfc/rfc6455.txt

[67] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Applied Cryptography and Network Security (ACNS 2011)*, J. Lopez and G. Tsudik, Eds. Berlin, Heidelberg: Springer, Jun. 2011, vol. LNCS 6715, pp. 1–18.

[68] M. G. Kendall, A. Stuart, and J. K. Ord, *Kendall's Advanced Theory of Statistics.* Oxford University Press (5th edition), 1991, vol. 2.

[69] Y. Wang, X. Li, J. Meng, Y. Zhao, Z. Zhang, and L. Guo, "Biprominer: Automatic mining of binary protocol features," in *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies.* Gwangju, South Korea: IEEE, Oct. 2011, pp. 179–184.

[70] Z. Yu, Y. Huang, S. Guo, B. Zhou, and H. Ren, "Extracting information from unknown protocols on campusnet," in *Proceedings of the 1st IEEE International Symposium on Information Technologies and Applications in Education.* Kunming, China: IEEE, Nov. 2007, pp. 535–539.

[71] S. Kleber, H. Kopp, and F. Kargl, "NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies*. Baltimore, Maryland, USA: USENIX Association, Aug. 2018.

[72] R. R. Sokal, "A statistical method for evaluating systematic relationships," *University of Kansas Scientific Bulletin*, vol. 38, pp. 1409–1438, Mar. 1958.

[73] J.-Z. Luo and S.-Z. Yu, "Position-based automatic reverse engineering of network protocols," *Journal of Network and Computer Applications*, vol. 36, no. 3, pp. 1070–1077, Feb. 2013.

[74] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo, "A semantics aware approach to automated reverse engineering unknown protocols," in *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP)*. Austin, Texas, USA: IEEE, Oct. 2012, pp. 1–10.

[75] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. Santiago de Chile, Chile: Morgan Kaufmann, Sep. 1994, pp. 487–499.

[76] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: Studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering*. Szeged, Hungary: ACM, Sep. 2011, pp. 448–451.

[77] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 592–597, Jun. 1972.

[78] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering*. Leipzig, Germany: ACM, May 2008, pp. 501–510.

[79] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*. New York, New York, USA: ACM, Aug. 2009, pp. 345–354.

[80] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* Szeged, Hungary: ACM, Sep. 2011, pp. 267–277.

[81] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference," in *Proceedings of the 29th ACM/IEEE International Conference on Automated software engineering.* Vasteras, Sweden: ACM, Sep. 2014, pp. 19–30.

[82] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering.* Hyderabad, India: ACM, Jun. 2014, pp. 468–479.

[83] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* Hong Kong, China: ACM, Nov. 2014, pp. 178–189.

[84] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[85] J. Yang and D. Evans, "Dynamically inferring temporal properties," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* Washington, D.C., USA: ACM, Jun. 2004, pp. 23–28.

[86] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, vol. 10, no. 1, pp. 27–52, Mar. 1978.

[87] J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *Proceedings of the 17th European Conference on Object-Oriented Programming.* Darmstadt, Germany: Springer, Jul. 2003, pp. 431–456.

[88] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* Lincoln, Nebraska, USA: IEEE, Jan. 2015, pp. 115–125.

[89] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Automated abstractions for contract validation," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 141–162, Nov. 2010.

[90] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Benevento, Italy: IEEE, Oct. 2006, pp. 51–60.

[91] D. Lo, S.-C. Khoo, and C. Liu, "Mining temporal rules for software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 227–247, 2008.

[92] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal api rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China: ACM, May 2006, pp. 282–291.

[93] D. Lo, S.-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. Seattle, Washington, USA: ACM, Jul. 2008, pp. 50–56.

[94] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of the International Conference on Computer Aided Verification*. Chicago, Illinois, USA: Springer, Jul. 2000, pp. 154–169.

[95] S. Schneider, I. Beschastnikh, S. Chernyak, M. D. Ernst, and Y. Brun, "Synoptic: Summarizing system logs with refinement," in *Proceedings of the 2010 Workshop on Managing systems via Log Analysis and Machine Learning Techniques*, Vancouver, British Columbia, Canada, Oct. 2010, pp. 1–10.

[96] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[97] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[98] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Providence, Rhode Island, USA: IEEE, Jul. 1977, pp. 46–57.

[99] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[100] D. Lo, L. Mariani, and M. Santoro, "Learning extended fsa from software: An empirical assessment," *Journal of Systems and Software*, vol. 85, no. 9, pp. 2063–2076, Sep. 2012.

[101] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog," in *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, California, USA: Citeseer, Feb. 2006, pp. 1–15.

[102] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*. Tucson, Arizona, USA: IEEE, Dec. 2005, pp. 203–214.

[103] C. Leita, M. Dacier, and F. Massicotte, "Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots," in *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Hamburg, Germany: Springer, Sep. 2006, pp. 185–205.

[104] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*. Raleigh, North Carolina, USA: ACM, Oct. 2012, pp. 37–48.

[105] M. A. Hossain, S. Versteeg, J. Han, M. A. Kabir, J. Jiang, and J.-G. Schneider, "Mining accurate message formats for service apis," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy: IEEE, Mar. 2018, pp. 266–276.

[106] J. Jiang, S. Versteeg, J. Han, M. A. Hossain, and J.-G. Schneider, "A positional keyword-based approach to inferring fine-grained message formats," *Future Generation Computer Systems*, vol. 102, pp. 369–381, Aug. 2019.

[107] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of Computational Biology*, vol. 1, no. 4, pp. 337–348, Jun. 1994.

[108] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, Jan. 2001.

[109] G. McLachlan, K.-A. Do, and C. Ambroise, *Analyzing Microarray Gene Expression Data*. Hoboken, New Jersey, USA: Wiley, 2004.

[110] S. Bapat, "Automatic storage of persistent ASN.1 objects in a relational schema," Mar. 1994, U.S. Patent 5,291,583.

[111] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1," May 2000. [Online]. Available: https://www.w3.org/TR/soap/

[112] Twitter Inc. (2014) Twitter REST API. [Accessed: Mar. 22, 2018]. [Online]. Available: https://developer.twitter.com/en/docs/api-reference-index

[113] Google LLC. Google Books API. [Online]. Available: https://www.googleapis.com/books/

[114] G. Combs *et al.*, "Wireshark," 2007. [Online]. Available: http://www.wireshark.org

[115] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 410–420.

[116] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Berlin Heidelberg, Germany: Springer Science & Business Media, 2008, p. 138.

[117] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[118] G. A. Babich and O. I. Camps, "Weighted parzen windows for pattern classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 5, pp. 567–570, May 1996.

[119] J. C. Bezdek and R. J. Hathaway, "VAT: A tool for visual assessment of (cluster) tendency," in *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN)*. Honolulu, Hawaii, USA: IEEE, May 2002, pp. 2225–2230.

[120] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, Jan. 2003.

[121] P. A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*. Upper Saddle River, New Jersey, USA: Prentice-Hall, 1982.