Service Virtualisation

Subjects: Others
Submitted by: Zeinab Farahmandpour

Definition

Continuous delivery is an industry software development approach that aims to reduce the delivery time of software and increase the quality assurance within a short development cycle. The fast delivery and improved quality require continuous testing of the developed software service. Testing services are complicated and costly and postponed to the end of development due to unavailability of the requisite services. Therefore, an empirical approach that has been utilised to overcome these challenges is to automate software testing by virtualising the requisite services’ behaviour for the system being tested. Service virtualisation involves analysing the behaviour of software services to uncover their external behaviour in order to generate a light-weight executable model of the requisite services. There are different research areas which can be used to create such a virtual model of services from network interactions or service execution logs, including message format extraction, inferring control model, data model and multi-service dependencies.

1. Introduction

Recently, software systems have been structured as several different components which communicate to accomplish software tasks. Due to some limitations, testing of these components may end up very costly or time-consuming. One of the limitations is the unavailability of other internal or external requisite components for testing any of components. This limitation can lead to a delay in the delivery of the components to the time when all of them are developed. This example illustrates one of the situations when all the requisite components are from one vendor. Using different vendors for the requisite components can cause the problem of testing to be even more severe.

Continuous delivery is one of the approaches in software engineering to deliver higher quality software faster [1]. Embracing continuous delivery reduces the integration cost which makes the continuous integration plausible. Continuous integration requires the continuous testing of the components every time they are either developed or changed. For the continuous testing of each component, all the requisite components need to be present regardless of their development status.

Several approaches aimed to provide the required components and environments ready for testing each component. The first approach is the commonly used mock objects such as stubs [2][3]. The server-side interactive behaviour of each requisite component is simulated in this approach by coding them in specific languages. This manual coding needs to be repeated every time each component is being modified.

Virtual machines tools such as VirtualBox and VMW are the second commonly used approach which aims to provide a platform to install multiple server systems [1]. This approach requires the availability of the server systems to be installed on the virtual machines, which may not be accessible in some situations. The need for the real service resource for the installation causes the method to suffer from scalability issues, which limits the number of systems that can be installed on one system.

Another approach to alleviating the high system resource demand in hardware virtualisation is container technologies such as Docker [1]. In comparison to other hardware virtualisation approaches, this solution requires fewer system resources and provides protected portions of the operating system; therefore, it can set up a lighter testing environment with higher scalability. However, this method still suffers from scalability limitations.

Service emulation [4][7] is a slightly recent approach which tries to replace each requisite component with
its behaviour estimation. This estimation is meant to be executable and very light-weight. This solution is used to target a specific component’s characteristics for the specific quality test and ignore the rest. Emulating each component in this approach requires the functionality information of each component and an expert’s manual configuration. The components’ functionality information may not be available in components with intricate behaviour.

To alleviate some of the service emulation limitations, the idea of service virtualisation (SV) was proposed. SV aims to get information about components’ behaviour directly from their network interactions. By utilising tools such as Wireshark, this method can record the components’ network interactions. Then, instead of human experts, it aims at using machine learning techniques to drive light-weight executable component models. Record-and-replay technique is another term used for the SV solution. This is because, in responding to a live request, the proposed SV tries to find the most similar request from those that have been recorded before and substitute some of its response fields to make it compatible with the new request. Figure 1 visualises the record-and-replay concept.

![Figure 1. Record and replay approach.](image)

The referenced SV solutions cover some simple stateless protocols that do not require more than the current request to generate an accurate response message. To accommodate stateful protocols—when the response message requires not only the current request but also the previous relevant messages—there are a few methods proposed by Enişer and Sen and Farahmandpour et al. However, the long short-term memory (LSTM) based method presented by Enişer and Sen to cover stateful protocols needs a long time and lots of resources to train.

These were the research efforts that directly contributed to the development and improvements of SV approaches. In contrast to the limited number of solutions within the SV field, there is a huge body of work that indirectly contributes to this area of the research. As a result, the rest of this paper provides an overview of different research efforts that can directly or indirectly contribute to the software virtualisation solution in a component based structure.

2. Related Work

Discovering or mining software/service behaviour turned into an area of focus in several domains of software engineering. As a result, several efforts have been devoted to the advancement of techniques to fulfill this. Software behaviour mining is concerned with the discovery of message formats, control, data
and multi-service dependency facets of software services from their execution logs or network interactions.

Service virtualisation provides a promising way to provision a realistic testing environment by modelling the approximate interactive behaviour of individual real software systems. Modelling of services can be derived using the service descriptions \(^{6,7}\) or other automatic techniques \(^{16}\) as a black-box approach. This black-box without looking into the service implementation or description by analysing the recorded interaction traces can achieve a similar outcome. Automatic service virtualisation includes analysing service interaction traces and deriving executable models for the services without requiring any knowledge about the services or their protocol message format. This process can lead to the emulation of a much wider range of protocols and services. This paper surveys existing methods of discovering models from a requisite service interacting with the service under the test (SUT). These methods analyse the SUT’s interaction traces and take into account the dependencies and mutual impacts between their traces.

SUT is interacting with the requisite service by sending and receiving network messages. The goal is to emulate/replicate the requisite service to generate the same messages, as is generated by the requisite service in response to the request messages sent by the SUT in a real environment. The generated response messages in virtual service need to satisfy three different aspects to be considered similar to a real service. The first aspect is the format of the messages, the second is having the same header information, and the third is containing the payload as expected. Virtualising requisite services can help with testing the SUT without connecting to the real requisite service and without the effort of specifying service descriptions.

The aim of virtualising a service is modelling the interactive behaviour of enterprise software systems to enable mimicking their run-time properties. Software services or components cooperate with each other through sending or receiving messages, regularly with the assumption that one service sends a request to other services, looking to receive a response (or multiple responses) in a short time period. Functional behaviours of virtualised services need to be precise as the service’s characteristics \(^2\).

There are four key aspects that need to be considered in order to mimic the full interactive behaviour of the requisite service. Firstly, the structure of messages have to be well-formed and adhere to their format in the protocol as they are generated by the actual requisite service, secondly, generating protocol-conformant header information \(^{16}\). This means sending service response messages that contain acceptable and expected header information such as response types. Thirdly, generating payload-conformant messages indicates that the content of the response message generated has to be identical with the actual payload. Last but not least is the multi-service dependencies which focus on control and data model dependencies between distributed services and try to incorporate them into the service’s behaviour.

As a result, the relevant literature is classified into four different areas, i.e., message format is discussed in Section 3, control model (temporal ordering) which is explained in Section 4, the data model will be reviewed in Section 5, and multi-service dependencies explained in Section 6. In the following sections, we introduce and analyse each area in detail.

3. Message Format
The message format is defined by the structural rules of forming the messages and their contents in a service’s interaction protocol or interface description. Every protocol may define several types of messages with each having its particular format. Understanding the protocol’s message format is crucial to mimic each service’s behaviour. Some of the researchers extract the key parts of messages and their offsets, which indicate their location from the start of the message, while others extract the full structure of messages. As Figure 2 shows, each message consists of a header and payload. Each header and payload can later be divided into their relevant bytes of information and tagged separately. This section
starts with the research dedicated to discovering partial message format and then continues reviewing work focused on inferring the whole message format.

![Figure 2. Message and its structural format.](image)

### 3.1. Summary of the message format

The related work surveyed above extracts the message structures used in a protocol. Some of them such as Du et al. [12][17][18], Cui et al. [19] and Comparetti et al. [20] cluster the messages based on their type in one cluster, and keep one common message as the template for the type. Then, they try to find the most important and dynamic parts of the messages that need to be changed in a new message using iterative clustering and/or statistical analysis such as frequency/entropy, TF/IDF, server's behaviour such as executed code fragments, library, system calls, output and file system activity, etc.

Others such as Cui et al. [21], Leita et al. [22], Krueger et al. [23], Small et al. [24], Wang et al. [25] and Jiang et al. [26][27] go further and try to infer the whole structure of the messages, field by field, using iterative clustering and statistical analysis of the messages to find the structures for each cluster. In addition, Small et al. [24] and Wang et al. [25] infer one FSM for each cluster. The existing binary message format extraction methods rarely extract the hierarchical structure of messages where the dependencies between different fields in a message are identified. In addition, some of them are designed to extract the structure of messages that have fixed-length fields and cannot work on protocols with the variable length fields or complex structures.

### 4. Control Model

This section discusses existing methods for extracting the temporal ordering or control model of services from their logs. These methods analyse the network interactions of service with other services or its internal execution logs as the only external recorded documents of the system’s behaviour and extracts the service's temporal behaviour in the form of a state machine. Following the state machine in these approaches, the next behaviour of the system is determined based on the previous behaviour of the system from the start of the session/interaction to the current point. Some other methods use deep learning methods to infer the next message or legitimate fields and values. The resulting control model for the service helps to identify the type of response messages for the incoming request messages.

#### 4.1. Summary of control model

Some of the approaches reviewed in this section extract a temporal behavioural model of a system as temporal invariants from their network interactions or internal execution logs or both of them. They use different temporal patterns to capture the system’s behaviour, including always occurrence, immediate occurrence, non-occurrence, etc. Each of the statistically significant constraints are kept as building blocks of FSA and others are considered as outliers. Moreover, most of the papers discussed consider the inferred invariants in the global scope, i.e., the whole event trace needs to match the inferred constraints.
There is also an example that defines a local region as the scope of the extracted pattern.

Existing studies try to mine and model services’ behaviour in the form of FSA based on two types of approaches. The first group, Comparetti et al. [20], Leita et al. [22], Wang et al. [25], Trifilò et al. [29], Antunes [30], and Kabir and Hun [10] build an FSA model from the logs (raw/pre-processed) and then try to abstract and generalize the model based on some heuristics or clustering methods. While Beschastnikh et al. [7], Yang and Evans [17], Lo et al. [23], and Le et al. [34] first infer invariants and refine them based on statistical techniques and then construct FSA. The papers tried to use a variety of methods to refine the FSA. They can be summarised as incrementally refining and merging likely equivalent states without violating the satisfied invariants, using counter-examples to split and refine the model, using domain-specific heuristics to identify similar states to merge, and bottom-up methods such as intra-trace refinement, which starts merging from each cluster of similar traces up to merging between different clusters called inter-trace merging.

As we can see, the existing literature infers a service’s temporal model based on its interaction behaviour with one service, without incorporating different types of dependencies to generate all the message payload required to virtualise a service’s behaviour. One service’s behaviour may be affected in different ways by another requisite service. Therefore, we need to expand the existing methods to consider the services’ mutual impacts on each other in generating both exact service response message types and their contents.

5. Data Models

This section surveys research efforts related to inferring data models from logged data. In some studies, data models are referred as data invariants. This section also reviews studies that add the inferred data models/data invariants to the temporal/control model.

The data invariants can help better understand software behaviour. Data dependency between two services can be considered at different levels of granularities. Intra-protocol data dependency are ones that concern data dependency between messages of one protocol. For example, at a more granular level, data dependency between the data fields of a request message and those of its corresponding response message are one type of inter-message dependency and part of the intra-protocol dependency. Such inter-message data dependency can help formulate a response message for a corresponding request message. We note that most of them utilise clustering to differentiate between different pairs of request-response messages.

5.1. Summary

Despite the advancement in the inference of the temporal model (control model), few researchers have addressed the problem of discovering data models. The above attempts tried to combine control models and some data invariant models in order to improve the inferred behavioural model. However, the coverage of the inferred data model in terms of the complexity of the model, the type, and the number of parameters are limited and can only reflect a limited range of data types and data dependency in the real environment. There remains a need to design and develop a more comprehensive dependency extraction method, which uses different methods of dependency discovery and different levels of complexity to infer complex data dependency models that describe the inter-dependency between systems.

6. Multi-Service Dependency

Looking to find the relevant work in other areas, such as multi-service dependency which could possibly be used to extract more complex dependency between services that can influence service behaviour has resulted in the following insights. As an example shown in Figure 3 taken from [35], we can see different dependencies that can influence the behaviour of service S3.
To extract the multi-service dependency model of distributed services, and get the total ordering of the events, the first step is the synchronisation of the services’ clock. In [36], Lamport examined the concept of one event happening before another in a distributed system and defined a partial ordering of the events. To synchronise a system of logical clocks, a distributed algorithm was introduced, which also can be used to totally order the events. Lamport then derived a bound on how far out of the synchrony the clock can become.

We categorise the related work below based on the type of data that they use to investigate the dependencies. Some of the studies use network-level data, some use application-level data, and others utilise both of them. Depending on criteria such as test goals, needs, and systems and resources access level, different methods are followed. Based on the captured data, they extract dependencies between distributed systems or components at different levels of granularities, and, in most cases, build a dependency graph to show the dependencies, their strength and direction of influence.

6.1. Summary

As we can see, the discussed methods try to extract the dependencies between distributed services or components. Some of them use network-level data, which use sending/receiving patterns in a time period, packet header data like (e.g., IP, port, protocol), delay distribution of the traffic, cross-correlation of data, equality of the attributes (key-based correlation) and reference-based correlation. Others leverage application-level data such as application ID and process ID, monitored data (e.g., resource usage, TCP/UDP connection number), and temporal dependency between the execution of systems, which allow them to have a better understanding of the systems’ behaviour and their dependencies.

The existing approaches try to extract simple explicit dependencies (i.e., dependencies between services with their direct traffic between them). Latent dependencies (where the dependencies cannot be identified at first glance and need mathematical calculation to discover) and implicit dependencies (data dependencies between distributed services without access to their traffic or through their traffic with other services) have only been covered in a very limited sense. For example, Oliner et al. [37] extracted part of latent dependencies using cross-correlation, but their effort only considers linear dependencies for the services that communicate directly in terms of sending and receiving messages. In general, each existing method tries to discover only one part of the simple dependencies in the real service’s traffic. Little attention has been paid to the selection of multiple and complex dependency functions that can

---

**Figure 3.** Multi-service dependency between different services [35].

---
identify latent dependencies or a combination of several dependencies. A major focus in existing service dependency discovery efforts has been on how to identify data-dependent services rather than incorporating their effects on the system’s behaviour. In general, there remains a need for efficient and systematic approaches that can cover all types of data dependencies between services and their mutual impacts on each other.

References


**Keywords**

service virtualisation; message format; control model; data model; multi-service dependencies