Testing Service Oriented Architectures Using Stateful Service Virtualization Via Machine Learning

Hasan Ferit Enişer
Dept. of Computer Engineering
Boğaziçi University, Turkey
hasan.eniser@boun.edu.tr

Alper Sen
Dept. of Computer Engineering
Boğaziçi University, Turkey
alper.sen@boun.edu.tr

ABSTRACT

Today's enterprise software systems are much complicated than the past. Increasing number of dependent applications, heterogeneous technologies and wide usage of Service Oriented Architectures (SOA), where numerous services communicate with each other, makes testing of such systems challenging. For testing these software systems, the concept of service virtualization is gaining popularity. Service virtualization is an automated technique to mimic the behavior of a given real service. Services can be classified as stateless or stateful services. Many services are stateful in nature. Although there are works in the literature for virtualization of stateless services, no such solution exists for stateful services. To the best of our knowledge, this is the first work for stateful service virtualization. We employ classification based and sequence-to-sequence based machine learning algorithms in developing our solutions. We demonstrate the validity of our approach on two data sets collected from real life services and obtain promising results.

CCS CONCEPTS

• **Software Testing**; • **Service Virtualization**; • **Machine Learning**;

KEYWORDS

Software Testing, Service Virtualization, Machine Learning

ACM Reference Format:
Hasan Ferit Enişer and Alper Sen. 2018. Testing Service Oriented Architectures Using Stateful Service Virtualization Via Machine Learning. In *AST'18: AST'18:IEEE/ACM 13th International Workshop on Automation of Software Test , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194733.3194737

1 INTRODUCTION

In enterprise software systems, Service Oriented Architectures (SOA) help companies achieve flexibility and scalability for business requirements. As a result of such architectures, today's enterprise software systems have higher number of interconnected services, interdependent teams and heterogeneous technologies. In such complex software systems, testers and developers would spend considerable amount of time to access a service because of the conditions below [25]:

AST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5743-2/18/05…$15.00
https://doi.org/10.1145/3194733.3194737

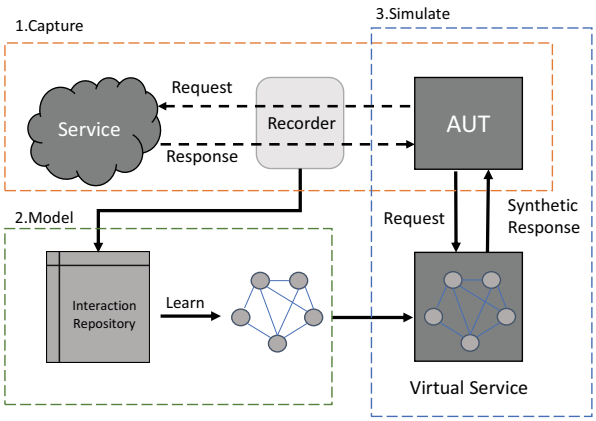

Figure 1: Overview of service virtualization.

- Still evolving or uncompleted services.
- Limited capacity or availability of services at inconvenient times.
- Services that are controlled by a third-party that grants restricted or costly access.
- Services that are needed simultaneously by different test teams with various setup and requirements.

Therefore software development requires *test doubles* that allow developers to decouple the application from its dependencies when testing the Application Under Test (AUT).

Service Virtualization is a practice to create virtual copies of a dependent service. Virtual services make testing in large software systems easier since they can be used to replace real services. They can simulate performance and data characteristics of the real service. Also, service virtualization is suitable for complex and very large legacy software that has many dependencies [25].

The fundamental process of service virtualization practice can be abstracted into three phases; capture, model and simulate as shown in Figure 1. First, the required information to virtualize a service (in terms of request-response messages) is captured, next a model is constructed using the captured data. This model correlates to the essence of the *virtual service*. Last, the virtual service is deployed to the development environment and simulated to return a synthetic response for a given request instead of the response of the real service. Services can be classified as stateless and stateful. In stateless services, all the information required for a correct response is supplied on the request. For example, a service that returns the list of hotels in a city is a stateless service. In stateful services, requests can change the state of the service and a response for

a given request is generated according to the current state. An example can be a shopping cart service where the service needs to remember the contents of the cart to correctly respond to requests that examine the cart, checkout, and so on. This class of services is harder to virtualize. Although, there are works on stateless service virtualization, to the best of our knowledge this is the first work on stateful service virtualization. Our main contributions in this paper are as follows:

- We propose techniques that directly attack to automated stateful service virtualization.
- We employ two machine learning techniques to obtain a virtual service model from captured request response pairs.
- We implement our techniques in a tool and validate our approach on real services.
- We compare our techniques with a state model inference tool.

The remainder of the paper is organized as follows. Section 2 describes the related work, section 3 gives the necessary background, section 4 details the technique that we introduced. Section 5 evaluates our approach, while Section 6 discusses the results and future work.

2 RELATED WORK

Service virtualization is a relatively new practice. In [25], the authors describe the basics of service virtualization. They also explain the benefits, capabilities and best practices of service virtualization. Nizamic et al. [28] present a case study of using virtual services for an enterprise health-care software. The latest studies on service virtualization are in [16, 17, 31, 33, 34], where the authors employ bioinformatics algorithms to virtualize stateless services. The latest one of these studies, namely [34] improves all previous results and presents a technique called Opaque Service Virtualization (OSV) for stateless services. Newly emerged containerization tools like Docker [6] are orthogonal to service virtualization. For example, a virtual service can be run in a container as well.

Leading software companies such as IBM, HP, CA, SmartBear, and Parasoft also provide various stateless commercial service virtualization tools. These tools are compared and evaluated in [19, 27]. A recent survey of service virtualization vendors conducted on real users can be found in [22].

We also surveyed studies in the field of black-box model inference that captures the state behavior of a software or a system. K-tails [5] is the most basic and well-known algorithm for black-box model inference. It serves as a basis for many model inference algorithms [14, 24, 30, 35]. Walkinshaw et al. [36] combine the information of event ordering and data values of events for model inference. They use data values to predict the next event by employing a classification algorithm. Dallmeir et al. [15] introduce ADABU to infer correct program behavior. They observe actual program executions to construct state machines, called *object behavior models*. Synoptic [4], CSight [3], and Perfume [29] use the CEGAR [11] approach to create a coarse initial state model, and then refine it using counterexamples that falsify temporal invariants. Krka et al. [23] group dynamic model inference strategies into four classes. They show that their technique in *trace-enhanced-invariants* class works best for several popular Java libraries. State machine extraction is also studied in neural networks field [8, 12, 18].

None of the above black-box model inference techniques is dedicated to predict the response of a specific request as we do in stateful service virtualization. Although conceptually one can reformulate the model inference problem in terms of a service virtualization problem (as we show for one such algorithm [36] in experiments), the cost of reformulating the problem is high and the performance of the algorithm is not satisfactory.

3 BACKGROUND

This section provides the background necessary for the rest of the paper. In subsection 3.1 we make necessary definitions and in subsection 3.2 we present an example service.

3.1 Definitions

Let $Req, Res, T_{req}, T_{res}, C_{Req}, C_{Res}$ be a finite set of requests, responses, request types, response types, request contents, response contents, respectively. A request, $req \in Req$ is a 2-tuple $(type, content)$, where $type \in T_{req}$ and $content \in C_{req}$. A response, $res \in Res$ is also a 2-tuple $(type, content)$, where $type \in T_{res}$ and $content \in C_{res}$. In this work, the type of a response can be either *success* or *fail*.

An interaction is defined as a request response pair: (req, res) with $req \in Req$ and $res \in Res$. We define an *interaction trace*, $it \in IT$ as a finite sequence of interactions observed during the execution of the service; $(req_1, res_1), (req_2, res_2), \ldots, (req_n, res_n)$. A Interaction Repository (IR) keeps all recorded interaction traces. Given an interaction trace it, we define the history, h, of a request req_i, as the following: $h_{req_i} = (req_1, res_1), \ldots, (req_{i-1}, res_{i-1}), (req_i)$, that is, the trace ends with req_i and the corresponding response is absent. Similarly, the *k history of a request* req_i is shown as: $h^k_{req_i} = (req_{i-k}, res_{i-k}), (req_{i-k+1}, res_{i-k+1}), \ldots, (req_i)$. The set of all histories for all requests in all interaction traces is denoted by H, whereas the set of all k histories of all requests in all interaction traces is denoted by H_k.

We divide the services into two classes; *stateless* and *stateful services*. A *stateful service*, $StatefulS : H \rightarrow Res$, is a function from the set of histories to the set of responses. Hence, given the history of a req_i we can determine its response $resp_i$. A stateless service, $StatelessS : Req \rightarrow Res$, on the other hand, is a function from the set of requests to the set of responses. The response of a request can solely be determined from the request itself. Informally, in a stateless service, a request's response is not dependent on the request's history. An example of a stateless service can be a service which returns the capital city of the given country in the request. We describe a sample stateful service in the next subsection.

A *virtual service*, $VS : H_k \rightarrow Res_{syn}$ is a function where H_k is k histories of all requests, Res_{syn} is a finite set of synthetic responses, where k is specified by the user. A synthetic response is an artificial response which is the same as the actual response in the perfect case. Since we focus on the virtualization of stateful services, from now on VS refers to a stateful service in this paper.

3.2 A Sample Stateful Service

We now provide a sample stateful calendar service where a client can create events, update events, delete events and get the latest information of events.

Consider the sample interaction trace shown in Figure 2. For this sample trace, requests are shown above the dashed lines with

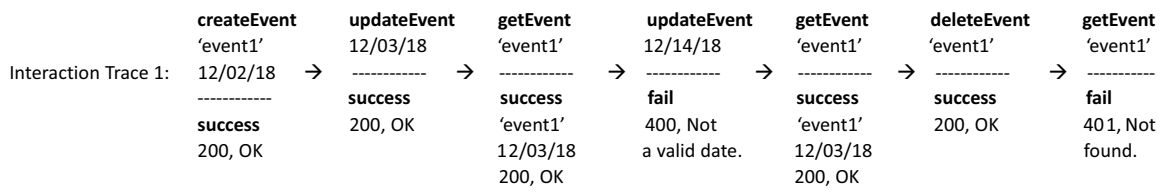

Figure 2: A sample interaction trace of a calendar service.

request types denoted in bold characters (*createEvent, updateEvent, deleteEvent* and *getEvent*) and the contents following the types.

Similarly, responses are shown below the dashed lines with request types denoted in bold characters (*success, fail*) and the response contents following the types.

According to the interaction trace, first, a calendar event was created, then its date was updated and then the details of the event was gathered (getEvent). Later, the user attempted to update the event again with an invalid parameter and received error. Thus, the event is not updated. After that, the user gets the event then deletes the event and tries to gather its information again. In this case, the service replies with an error since the event is deleted whereas earlier getEvent was successful.

This trace shows that in a stateful service to predict the correct response, the history of a request has to be considered instead of only the current request. This is because the current request's response can be affected by one of the previous events (deleteEvent).

4 METHOD

In this section, we introduce two different approaches for stateful service virtualization by learning the function VS defined earlier. We want to simulate the response behaviour of a service for a given request, as discussed above. For this simulation, we analyze the recorded messages and construct a model. Then, we make predictions based on the constructed model. We construct models using classification and sequence-to-sequence learning based techniques. We define response contents as classes to be predicted and requests as features in the classification based technique. In sequence-to-sequence based technique we learn the translation from requests to responses.

4.1 Classification Based Virtualization (CBV)

We first use classification based approach for service virtualization. Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output [2]. An example classification problem can be assigning customers to two classes: low-risk and high-risk. The information about a customer such as the income and savings form the input to the classifier whose task is to map the input to one of the two classes.

In a stateful service, a request's response is affected by previous interactions in the history. Therefore, we have to train a model that learns for each request req_i, the mapping between the history h_{req_i} and the response res_i . Specifically, we use the k history of requests and vary k for finding the best performing model. In real life services, mostly both h_{req_i} and res_i show characteristics of categorical data, thus we employ one-hot encoding in our approach.

Figure 3 shows the inputs and the outputs corresponding to $h^2_{req_3}$ and C_{res_3} in the interaction trace in Figure 2, where req_3 corresponds to the request of type *getEvent* and $k = 2$. Note that we use one-hot encoding for our input data, whereas we enumerate the prediction of the classifier, where the prediction is the response of the request. This is because in classification, outputs are only class labels, therefore ordinality is not the case. Hence, before training a model, we keep a set of all response contents. For the interaction trace in Figure 2, this set contains *200 OK, 400 Not a valid date, 401 Not found, 12/03/2018* and *event1*. Outputs [4, 5, 1] in Figure 3 are enumerated values of the elements in this set.

If there is more than one class to be predicted and those classes can possibly be assigned to more than two types of labels, this is called *multioutput-multiclass classification* [7]. The datapoint in Figure 3 has three outputs: the event's label, the event's date, and the result type. Also, each field has various number of possible values to take.

We employ Repeated Incremental Pruning to Produce Error Reduction (RIPPER) which is a pure rule based classification algorithm proposed in [13]. RIPPER produces a set of IF-THEN rules for separation. Note that, response prediction can be handled by defining appropriate decisions. An example decision would be, *if the response of updateEvent request is not 200, OK then ignore it.* Therefore, a rule based classification technique perfectly fits in response prediction in this case. Other advantages of using RIPPER include interpretability of the resulting rules and shorter training times.

After training the classifier, it is ready to predict responses for a given request. When a request arrives, we encode its history and give the encoding as the input to the classifier. If the incoming request or the history contains a feature that is not seen in training data, it is encoded in a way that is different from all other features in the training data. The classifier predicts the corresponding response. Also note that, this technique requires parsing the interactions to find request types, contents and the response to be encoded. Therefore, it is more appropriate to use this technique on services communicating in well-known message formats such as JSON or XML. This is because each service with custom message format requires a new parser. Also, parsing of custom message formats can be painful.

4.2 Sequence-to-Sequence Based Virtualization (SSBV)

In this section, we describe another approach for virtualization of stateful services. We train a *sequence-to-sequence* model to learn the function VS for this approach. This technique, can be used easily for services that use custom message formats for performance and security purposes, since it does not require parsing unlike CBV.

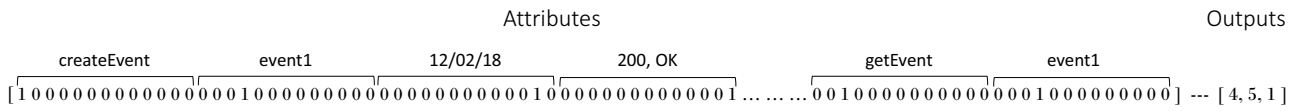

Figure 3: An example datapoint provided for training of Classification Based Virtualization.

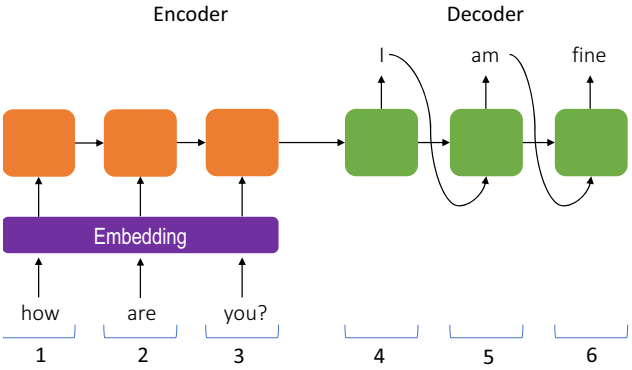

Figure 4: The general outline of sequence-to-sequence models consisting of an encoder and a decoder. The sequence *how are you?* is transformed to sequence *I am fine.*

Sequence-to-sequence models were previously employed in problems requiring to consider whole history of input such as language translation [32] or automatic chat-bot creation [37, 38] and demonstrated successful results. *Sequence-to-sequence* models use a special form of *Recurrent neural networks* (RNNs), namely, *Long Short Term Memory* [21]. LSTMs allow the usage of historical data in several steps in the future. This is crucial because stateful services use historical data. The basic architecture of a sequence-to-sequence model is depicted in Figure 4. The architecture consists of two LSTM networks [9]; an *Encoder* (with embedding phase) that processes the input sequence and a *Decoder* that produces the output sequence, which are both LSTM networks. We use Tensorflow [1] for sequence-to-sequence learning.

We start by creating a *vocabulary* from the corpus, *IR* in our case. Previous works [32, 37] add each unique word in the corpus to the vocabulary since their aim is to find the correspondences between the words and the sentences. In our case we assume that we can not parse the interactions (they come from a custom format). Therefore, we create a vocabulary with the letters and characters in *IR*. Then the elements in the vocabulary are enumerated. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input. The transformed input is given to an encoder network. The encoder learns to encode an input sequence into a vector and the decoder learns to decode this vector back to the output sequence.

In training of our sequence-to-sequence model, we use prefixes of interaction traces, hence the model is learned iteratively. For example, Table 1 represents the inputs and the outputs for interaction trace 1 of Figure 2. Our experience in this work showed that this way of training makes the model learn the mappings faster.

5 EVALUATION

We implement both CBV and SSBV approaches to virtualize real stateful services. We also compare them with the EFSM inference technique described in [36], which aims to infer extended finite state machines from traces for reverse engineering. In this technique, during state machine construction, states are merged iteratively by predicting the next event using classification algorithms. We reformulate it to solve service virtualization problem.

5.1 Subject Services

We used two services in evaluation, namely, a proprietary Airline Ticketing Service (ATS) and an open source Google Calendar API (Calendar). Calendar uses the JSON format, whereas ATS uses a custom format for messaging, for which we implemented a parser.

Interactions collected from ATS corresponds to previous ticketing operations for testing purposes. For Calendar, we implemented a script in Python that sends random requests to the API and saves the responses. We collected 400 traces each with 10 interactions for these particular services. Our experiences showed that 400 traces are adequate for training of both CBV and SSBV methods. We reached this number by experimenting with the data. We also note that ATS data includes 26 different request types, whereas Calendar API data includes 5 request types.

5.2 Evaluation Setup

5.2.1 Metrics. We evaluated the techniques proposed in this work in terms of performance and correctness. Performance refers to the training time of the models. For correctness of CBV, we calculate two correctness scores, namely, exact matching ratio (accuracy) and subset matching ratio [26]. For correctness of SSBV, we use accuracy. There is a difference in the measures for CBV and SSBV because CBV predicts response classes, whereas SSBV predicts the responses themselves.

We now define the metrics used in correctness. Let E be the expected outputs, P be the predicted outputs, n the number of tests in validation phase, and p the number outputs predicted for each test, (which is three in the example datapoints shown in Figure 3 (e.g. 5, 4 and 1)). The set $C = \{c_1, c_2, \ldots, c_n\}$ is the set of all classes to be predicted. There are five different classes (200 OK, 400 Not a valid date, 401 Not found, 12/03/18, event1) to be predicted for the getEvent request shown in interaction trace in Figure 2. $\mathbb{1}$ denotes a slightly modified version of indicator function. $\mathbb{1}$ of a set A and set X is a function

$$\mathbb{1}_A : X \to \{0,1\} \quad \text{defined as} \quad \mathbb{1}_A(x) := \begin{cases} 1, & \text{if } x = A \\ 0, & \text{if } x \neq A \end{cases}$$

$$ExactMatchRatio = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{E_i}(P_i)$$

Clearly, a disadvantage of this measure is that it doesn't distinguish between complete incorrect and partially correct predictions which can be considered as harsh for some cases. Therefore we define subset matching ratio:

$$SubsetMatchRatio = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p} \mathbb{1}_{E_{ij}}(P_{ij})$$

Table 1: Part of the inputs and the outputs corresponding to interactions in Interaction Trace 1 used to train our sequence-to-sequence model. Spaces are put for clarity here, for actual training we do not put spaces if it is not included in data.

Input	Output
createEvent event1	200, OK
createEvent event1 12/02/18 200, OK updateEvent 12/03/18	200, OK
createEvent event1 12/02/18 200, OK updateEvent 12/03/18 200, OK getEvent event1	event1 12/03/18 200, OK

Higher SubsetMatchRatio values are better. We also calculate macro and micro averaged f-scores for evaluation of CBV method. In macro average, metrics are calculated for each label, and their unweighted mean is found. In micro average, metrics are calculated globally. F-score is originally defined for binary classification, however, an extension is proposed for multi-label classification. We modify macro and micro averaged f-score calculations proposed for multi-label classification [26] as follows:

$$F^c_{macro} = \frac{2\sum_{i=1}^{n}\sum_{j=1}^{p} Y^c_{ij} Z^c_{ij}}{\sum_{i=1}^{n} Z^c_i + \sum_{i=1}^{n} Y^c_i} \quad F_{macro} = \frac{1}{|C|}\sum_{k=1}^{|C|} F^{c_k}_{macro}$$

$$F_{micro} = \frac{2\sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Y^{c_k}_{ij} Z^{c_k}_{ij}}{\sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Z^{c_k}_{ij} + \sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Y^{c_k}_{ij}}$$

where

$$Y^{c_k}_{ij} = \begin{cases} 1, & \text{if } c_k \text{ is actually at } Y_{ij} \\ 0, & \text{otherwise} \end{cases}$$

$$Z^{c_k}_{ij} = \begin{cases} 1, & \text{if } c_k \text{ is correctly predicted at } Z_{ij} \\ 0, & \text{otherwise} \end{cases}$$

In SSBV we use *accuracy* for evaluation. Accuracy in SSBV is defined as follows. If for a given request history, the output of the real service and the output of the corresponding virtual service are the same, then the response predicted by the virtual service is an accurate response.

5.2.2 Research Questions. We describe the research problems that we evaluated with the results of our experiments. We look for the answers of the research questions given below:
RQ1: How successful are the techniques introduced in this work in replacing real services in terms of correctness?
RQ2: How effective are the techniques introduced in this work in terms of performance?

5.2.3 Experimental Design. Parameters selected in experiments are depicted in Table 2. Our primary concern in choosing those parameters is maximizing the correctness of the models.

For each technique, we created virtual service models with history sizes of $k \in 1, 5, 10$. For $k = 1$, we consider the last interaction, and for $k = 5$ ($k = 10$) we consider the last 5 (10) interactions including the incoming request.

For CBV method, we use Weka [20] implementation of RIPPER. Weka is a library of machine learning algorithms for data mining tasks. Also, for CBV experiments, we implemented a parser for ATS and we employed Python's built-in JSON parser for Calendar. For SSBV method, we created models using Tensorflow [1] implementation of sequence-to-sequence model with history sizes $k \in 1, 5, 10$. We employed Keras [10] for running Tensorflow framework which is a high-level neural-network API running on top of Tensorflow. We downloaded the EFSM tool [36]. For EFSM, we used J48 option [20] for classification. This option generates C4.5 decision trees. We reformulated the model inference problem in terms of a service virtualization problem. In our reformulation of EFSM tool, the next response is predicted according to previous interactions as well. However, since EFSM predicts one response content at a time, we trained more than one EFSM model.

Table 2: Parameters selected in experiments.

	CBV
RIPPER	*minNo = 1*
	SSBV
Tensorflow	*hidden_size = 128* *batch_size = 128* *layers = 2* *epochs=1* *iteration = 1000*
	EFSM Tool
J48	*default*

In the experiments, for each service, we performed three steps: (1) collect interactions from real service, (2) learn models using the collected interactions, and (3) measure the correctness and performance. We employed 5-fold cross validation for CBV and EFSM Tool where we use 80% of the data for training and 20% of the data for validation for each fold. Experiments were run on a server with 16 GB memory and Intel Xeon E5-2630L v2 2.40GHz CPU.

5.3 Evaluation Results

We now present and discuss the results of the experiments. First, we evaluate correctness and then we evaluate performance of the methods developed in this paper. We present the results of the experiments in Table 3 and Table 4. The results demonstrated in these tables are achieved with parameters shown in Table 2.

5.3.1 RQ1: Correctness Results. We achieve the highest accuracy, exact matching ratio and subset matching ratio with a history size of 10 for both ATS and Calendar API. We observed that matching ratios obtained from CBV method are slightly higher than those for EFSM Tool. This similarity is expected since both are classification based techniques. However, EFSM Tool does not consider the history while predicting a response. Therefore, we obtain higher values with CBV for most of the experiments. Also, F_{micro} and F_{macro} scores support results achieved in exact and subset match ratios. The highest F_{micro} and F_{macro} are close to 0.9 which is satisfying for correctness.

We also employed statistical tests to learn if the difference in correctness between CBV and EFSM Tool are significant or not. First,

Table 3: Correctness results of CBV, SSBV and EFSM Tool. EMR stands for Exact Matching Ratio, SMR stands for Subset Matching Ratio.

Service	k	CBV				EFSM Tool				SSBV
		EMR(%)	SMR(%)	F_{macro}	F_{micro}	EMR(%)	SMR(%)	F_{macro}	F_{micro}	Accuracy
ATS	1	76.6	79.6	0.781	0.767	78.1	80.6	0.803	0.783	92.1
Calendar	1	70.3	78.5	0.757	0.741	70.7	76.0	0.721	0.715	93.2
ATS	5	82.7	84.3	0.813	0.798	80.0	83.7	0.806	0.786	96.5
Calendar	5	81.1	84.1	0.843	0.825	71.5	77.1	0.753	0.741	97.3
ATS	10	82.7	84.3	0.813	0.798	80.0	83.7	0.806	0.786	96.5
Calendar	10	82.0	88.5	0.871	0.866	72.1	78.0	0.767	0.751	99.3

Table 4: Performance results of CBV, SSBV and EFSM Tool. Training time in format (hh:mm).

Service	k	SSBV	CBV	EFSM Tool
ATS	1	01:42	00:01	00:06
Calendar	1	02:21	00:01	00:06
ATS	5	08:51	00:03	00:11
Calendar	5	09.54	00:03	00:14
ATS	10	14:42	00:04	00:18
Calendar	10	16:19	00:04	00:19

we applied Shapiro-Wilk test to learn if our samples are distributed normally. Our samples consist of EMR and SMR results and f-score ratios of each fold in our 5-fold cross validation process. We consider only k=10 history of correctness results which are the highest among all k values. According to the results of Shapiro-Wilk test our samples are distributed normally for EMR, SMR and f-score metrics for CBV and EFSM Tool. Since normality assumption is achieved, we performed one sided two sample t-test to check whether the difference in results between CBV and EFSM Tool are statistically significant or not. Results showed that true difference in correctness is not equal to 0 with 95% confidence interval for EMR, SMR and f-scores. This means that EMR, SMR and f-score results are higher for CBV compared to EFSM Tool with 95% confidence. Note that these statistical tests are performed for both of ATS and Calendar API data separately.

Results show that the most successful technique in terms of correctness is SSBV for both ATS and Calendar services. Virtual services created using SSBV technique with history size of 10 behave almost the same as ATS or Calendar.

When we investigate the data in detail, we observed that there can be responses which are not possible to predict using automated methods like CBV or SSBV. An example of such a response can be a response including today's date which is not possible to predict using CBV or SSBV. However, these kind of interactions can be handled with a small manual effort in practice. Also we observe that, we achieve higher match ratios for ATS when compared to Calendar API. This is because ATS has limited type of request and response contents, however in Calendar there are extensive types of request and response contents.

In summary, we find that virtual services created using SSBV technique are accurate enough to replace the real services when 90% or more accuracy is needed. Virtual services created using CBV technique can replace the real services when an exact match is not required and a high subset match is enough.

5.3.2 RQ2: Performance Results. In CBV method, virtual services are created much faster than the SSBV technique. The training time of RIPPER in CBV method is in minutes, whereas, the training time of SSBV takes hours. Note that we conducted our experiments on a CPU and deep learning algorithms can run faster with GPUs. The training time of the EFSM tool is shorter than SSBV method, but longer than CBV method. If time is not important SSBV method can be used to virtualize a service since SSBV is the most successful method for generating correct responses. If time is limited it would be logical to use CBV method instead of EFSM Tool since again correctness in CBV is higher than EFSM Tool and also CBV runs faster. One advantage of EFSM Tool over CBV and SSBV is the model inferred. EFSM tool is originally designed for reverse engineering [36] and can extract behavioral models of the systems as EFSMs.

5.4 Threats to Validity

In this section, we discuss threats to our evaluation's validity.

Internal validity. The selected services in this work do not imply any bias. They are taken from different business areas and also their message format are different from each other.

Interactions from Calendar API are, in part, manually collected such that the parameters to requests are manually specified. This may bias the evaluation results. To mitigate this threat, we created a set for each parameter field and randomly chose one element from each set for each request. This makes a huge number of combinations of parameters for each request and prevent any bias for the results. Interactions from ATS are collected during testing of a dependent service, therefore we do not expect ATS data bias the results.

Some of the requests in ATS have only one type of response. Thus all the techniques mentioned in this work can predict correct responses for such requests which makes overall correctness higher. However, ATS is used in real life. This demonstrates us that there can be such cases in the industry. Therefore we do not expect that this threats our validity.

The algorithms used in this work can produce different results according to the parameters given. We use the best achieving parameters for all tools to prevent any bias in evaluation.

External Validity. Subject services in this work are selected from real life services. Additionally, the custom message format of

ATS and the standard JSON message format of Calendar API make a good combination in terms of evaluation of techniques. Also, the techniques proposed in this work can easily be implemented. Thus, we say that our results are generalizable. However, these approaches would not work if interactions are encrypted.

Construct Validity. The metrics specified in this work are performance and correctness which are the main concerns in service virtualization. We measure the correctness of SSBV with accuracy. On the other hand, we define exact matching and subset matching ratios for CBV to evaluate its success in detail. Also, we measure micro and macro averaged f-scores, since sometimes raw results of classification do not tell the whole story.

6 CONCLUSION

In multi-layered and service oriented architecture based software systems, testing and development can be painful because of interdependent nature of such systems. This paper studied how to overcome the dependency issue by creating virtual services that can be used during testing. To the best our knowledge, this is the first work on stateful service virtualization. We introduced two novel machine learning based methods to create virtual services. Our first technique (CBV) transforms response prediction problem into a classification problem and our second technique (SSBV) employs sequence-to-sequence models from deep learning.

Our evaluation demonstrates that virtual services trained using CBV technique significantly outperform other methods in terms of training time. On the other hand, virtual services trained using SSBV model produce the most accurate responses. Our results highlight the importance of the trace size in training such that longer traces result in more accurate responses. We also observe that methods proposed in this work are better in predicting responses than the technique presented in [36], which is originally designed to infer state models for reverse engineering.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Ethem Alpaydin. 2014. *Introduction to machine learning*. MIT press.

[3] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *International Conference on Software Engineering*.

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ACM SIGSOFT symposium and European conference on Foundations of software engineering*.

[5] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers* 100, 6 (1972), 592–597.

[6] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015).

[7] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*.

[8] Mike Casey. 2008. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Dynamics* 8, 6 (2008).

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[10] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras. (2015).

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer aided verification*. Springer.

[12] Axel Cleeremans, David Servan-Schreiber, and James L McClelland. 1989. Finite state automata and simple recurrent networks. *Neural computation* 1, 3 (1989), 372–381.

[13] William W. Cohen. 1995. Fast Effective Rule Induction. In *12th International Conference on Machine Learning*.

[14] Jonathan E Cook and Alexander L Wolf. 1998. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 3 (1998), 215–249.

[15] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *International workshop on Dynamic systems analysis*.

[16] Miao Du, Jean-Guy Schneider, Cameron Hine, John Grundy, and Steve Versteeg. 2013. Generating service models by trace subsequence substitution. In *International ACM Sigsoft conference on Quality of software architectures*.

[17] Miao Du, Steve Versteeg, Jean-Guy Schneider, Jun Han, and John Grundy. 2015. Interaction traces mining for efficient system responses generation. *SIGSOFT Software Engineering Notes* 40, 1 (2015).

[18] C Lee Giles, Clifford B Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. 2008. Learning and extracting finite state automata with second-order recurrent neural networks. *Learning* 4, 3 (2008).

[19] Diego Lo Giudice. 2014. Service Virtualization And Testing Solutions. *Forrester Wave* (2014).

[20] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11, 1 (2009), 10–18.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[22] IT-Central-Station. 2017. Service Virtualization A Peek Into What Real Users Think. https://goo.gl/oN23qH. (2017). accessed at December 2017.

[23] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 178–189.

[24] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *International conference on Software engineering*. ACM, 501–510.

[25] John Michelsen and Jason English. 2012. What is service virtualization? In *Service Virtualization*. Springer, 27–35.

[26] S Sorower Mohammad. 2010. A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis* 73 (2010).

[27] Thomas E Murphy and Nathan Wilson. 2013. Magic Quadrant for Integrated Software Quality Suites. *Gartner Research* (2013).

[28] F Nizamic, R Groenboom, and A Lazovik. 2011. Testing for highly distributed service-oriented systems using virtual environments. *Dutch Testing Day* (2011).

[29] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral resource-aware model inference. In *International conference on Automated software engineering*.

[30] Steven P Reiss and Manos Renieris. 2001. Encoding program executions. In *International Conference on Software Engineering*.

[31] Jean-Guy Schneider, Peter Mandile, and Steve Versteeg. 2015. Generalized Suffix Tree based Multiple Sequence Alignment for Service Virtualization. In *Australasian Software Engineering Conference (ASWEC)*.

[32] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*.

[33] Steve Versteeg, Miao Du, John Bird, Jean-Guy Schneider, John Grundy, and Jun Han. 2016. Enhanced playback of automated service emulation models using entropy analysis. In *International Workshop on Continuous Software Evolution and Delivery (CSED)*.

[34] Steve Versteeg, Miao Du, Jean-Guy Schneider, John Grundy, Jun Han, and Menka Goyal. 2016. Opaque service virtualisation: a practical tool for emulating endpoint systems. In *International Conference on Software Engineering Companion*.

[35] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *International Conference on Automated Software Engineering*.

[36] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.

[37] Anbang Xu, Zhe Liu, Yufan Guo, Vibha Sinha, and Rama Akkiraju. 2017. A New Chatbot for Customer Service on Social Media. In *Conference on Human Factors in Computing Systems*.

[38] Xiaoqiang Zhou, Baotian Hu, Qingcai Chen, Buzhou Tang, and Xiaolong Wang. 2015. Answer sequence learning with neural networks for answer selection in community question answering. *arXiv preprint arXiv:1506.06490* (2015).