Testing Service Oriented Architectures Using Stateful Service Virtualization Via Machine Learning

Hasan Ferit Enişer, Alper Sen

{hasan.eniser,alper.sen}@boun.edu.tr
depend.cmpe.boun.edu.tr

Department of Computer Engineering
Boğaziçi University

AST 2018

Table of Contents

1. Introduction
2. Related Work
3. Method
4. Evaluation
5. Conclusions

Table of Contents

1 Introduction

2 Related Work

3 Method

4 Evaluation

5 Conclusions

Motivation

In enterprise software systems, Service Oriented Architectures (SOA) help companies to achieve flexibility and scalability for business requirements.

Motivation

In enterprise software systems, Service Oriented Architectures (SOA) help companies to 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.

Motivation

In such complex software systems, testers and developers suffer from the conditions below:

- Still evolving or uncompleted services.

Motivation

In such complex software systems, testers and developers suffer from the conditions below:

- Still evolving or uncompleted services.
- Limited capacity or availability of services at inconvenient times.

Motivation

In such complex software systems, testers and developers suffer from the conditions below:

- 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.

Motivation

In such complex software systems, testers and developers suffer from the conditions below:

- 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 set up and requirements.

Motivation

In such complex software systems, testers and developers suffer from the conditions below:

- 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 set up and requirements.

Service virtualization can address some of these conditions.

Service Virtualization

Service Virtualization is a practice to create a virtual copy of a dependent component.

Service Virtualization

Service Virtualization is a practice to create a virtual copy of a dependent component.

Service virtualization:

- is suitable for sharing within a team and across teams.

Service Virtualization

Service Virtualization is a practice to create a virtual copy of a dependent component.

Service virtualization:

- is suitable for sharing within a team and across teams.
- is suitable for complex and very large (legacy) software that has many dependencies.

Service Virtualization

Service Virtualization is a practice to create a virtual copy of a dependent component.

Service virtualization:

- is suitable for sharing within a team and across teams.
- is suitable for complex and very large (legacy) software that has many dependencies.
- can simulate performance and data characteristics of the real component.

Service Virtualization

Service Virtualization is a practice to create a virtual copy of a dependent component.

Service virtualization:

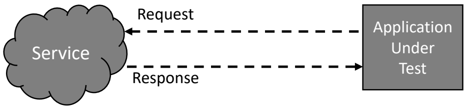
- is suitable for sharing within a team and across teams.
- is suitable for complex and very large (legacy) software that has many dependencies.
- can simulate performance and data characteristics of the real component.
- is useful for *test data management*.

Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.

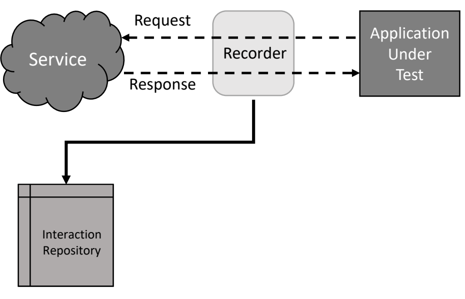
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



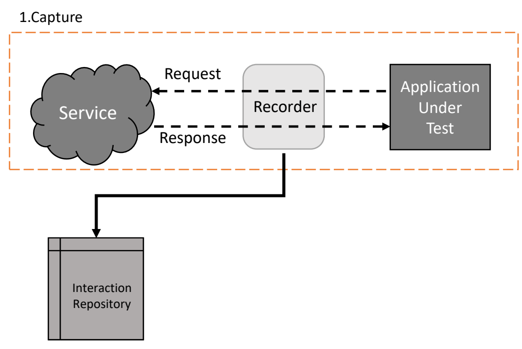
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



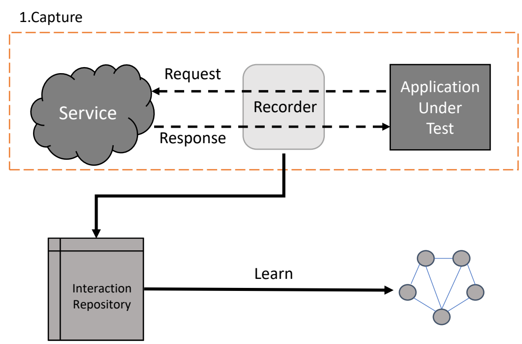
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



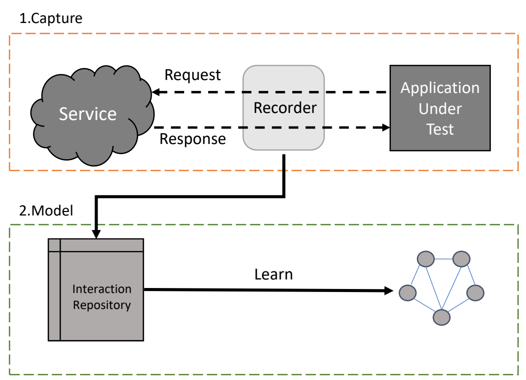
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



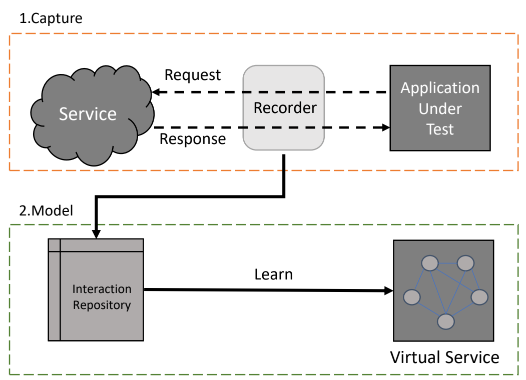
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



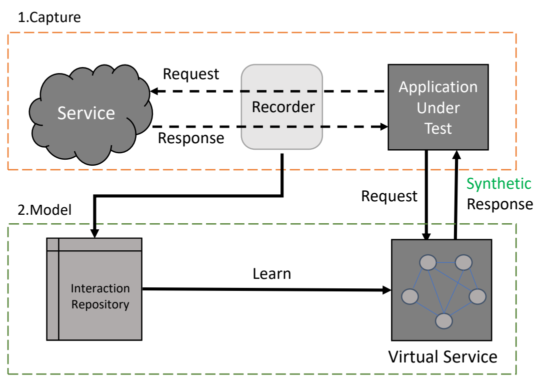
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



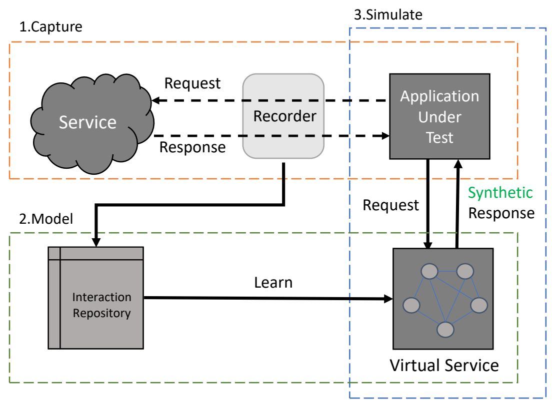
Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



Service Virtualization

The fundamental process of service virtualization practice can be abstracted into three phases; **capture**, **model** and **simulate**.



Service Virtualization

Services can be examined in two groups:

- Stateless services and
- Stateful services.

Service Virtualization

Services can be examined in two groups:

- Stateless services and
- Stateful services.

Stateful services require to keep state history to predict the response of a request.

Service Virtualization

Services can be examined in two groups:

- Stateless services and
- Stateful services.

Stateful services require to keep state history to predict the response of a request.

An example stateful service can be a shopping cart service where a user must login first to get cart information.Actions like logging in, adding or removing items from the cart bring the service to a new state.

Service Virtualization

Services can be examined in two groups:

- Stateless services and
- Stateful services.

Stateful services require to keep state history to predict the response of a request.

An example stateful service can be a shopping cart service where a user must login first to get cart information.Actions like logging in, adding or removing items from the cart bring the service to a new state.

Another example can be calender service where a user can create, delete or get events with a specific label. The user also can update event information.

Stateful Services

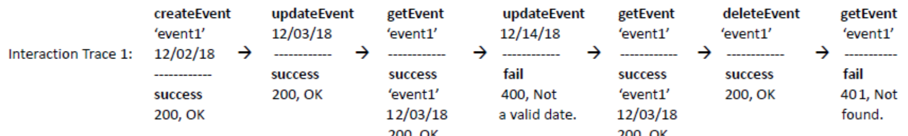


Figure: A sample interaction (request-response pair) trace.

Contributions

In this work,

- We propose techniques for automated *stateful service virtualization*.

Contributions

In this work,

- We propose techniques for automated *stateful service virtualization*.
- We employ two machine learning techniques to obtain a virtual service model from captured request response pairs.

Contributions

In this work,

- We propose techniques for 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.

Contributions

In this work,

- We propose techniques for 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.

Table of Contents

1 Introduction

2 Related Work

3 Method

4 Evaluation

5 Conclusions

Related Work

Leading software companies such as IBM, HP, CA, SmartBear, and Parasoft provide various commercial service virtualization tools. These tools are compared and evaluated in reports [4, 5].

Current service virtualization solutions in the literature [1, 2, 6, 7] have limited accuracy and performance and they are applicable to stateless services only.

Our previous work also provides a solution for stateless service virtualization. [3]

As far as we know, there is no previous study tackling with stateful service virtualization.

Table of Contents

1 Introduction

2 Related Work

3 Method

4 Evaluation

5 Conclusions

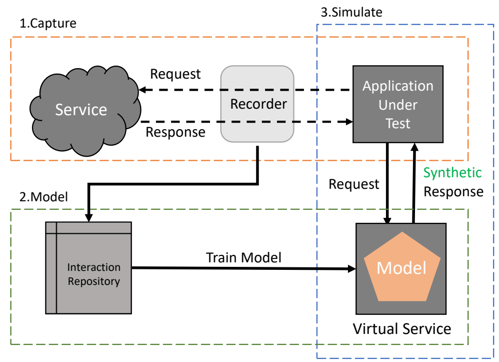
Proposed Techniques

We introduce two approaches for stateful service virtualization.

In the first technique named **Classification Based Virtualization (CBV)**, we formulate the response generation problem into a classification problem.

In the second technique named **Sequence-to-Sequence Based Virtualization (SSBV)**, we employ sequence-to-sequence models, which is a deep learning algorithm used in transformation of sequences from one form to another form.

Stateful Service Virtualization



Classification Based Virtualization (CBV)

Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output.

Classification Based Virtualization (CBV)

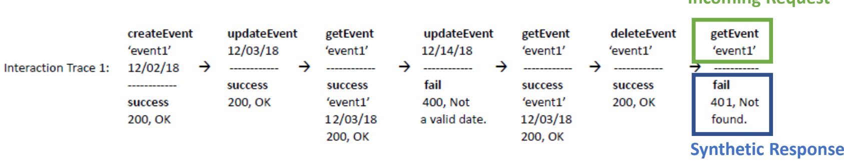
Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output.

In a stateful service, a request's response is affected by previous interactions in the history.

Classification Based Virtualization (CBV)

Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output.

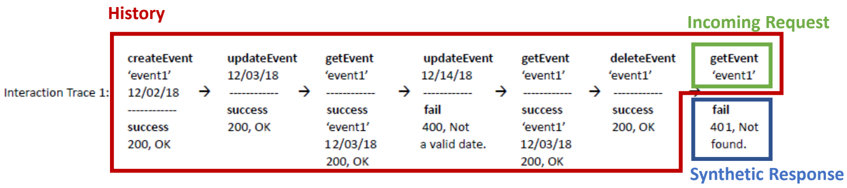
In a stateful service, a request's response is affected by previous interactions in the history.



Classification Based Virtualization (CBV)

Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output.

In a stateful service, a request's response is affected by previous interactions in the history.



Therefore, we train a classifier that learns the mapping between the history of requests and corresponding responses.

Classification Based Virtualization (CBV)

Inputs of this model show characteristics of categorical data, thus we employ *one-hot encoding* in our approach.

Classification Based Virtualization (CBV)

Inputs of this model show characteristics of categorical data, thus we employ *one-hot encoding* in our approach.

Figure: An example datapoint that is provided to the classifier.

Classification Based Virtualization (CBV)

Inputs of this model show characteristics of categorical data, thus we employ *one-hot encoding* in our approach.

Attributes | Outputs

createEvent | event1 | 12/02/18 | 200, OK | getEvent | event1

[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 ... ... 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0] --- [1, 2, 4]

Figure: An example datapoint that is provided 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.

Classification Based Virtualization (CBV)

We obtain best results with Repeated Incremental Pruning to Produce Error Reduction (RIPPER) which is a pure rule based classification algorithm. RIPPER produces a set of IF-THEN rules for separation.

Note that, we predict more than one class and those classes can possibly be assigned to more than two types of labels. This kind of classification is called *multioutput-multiclass classification*.

This technique requires parsing the interactions to find request types, contents and the response to be encoded. Appropriate to use it on well-known message protocols e.g. JSON, XML.

Sequence-to-Sequence Based Virtualization (SSBV)

A sequence-to-sequence model is a special form of Recurrent Neural Network (RNNs), namely, Long Short Term Memory (LSTM). LSTMs allow the usage of historical data in several steps in the future.

Sequence-to-Sequence Based Virtualization (SSBV)

A sequence-to-sequence model is a special form of Recurrent Neural Network (RNNs), namely, Long Short Term Memory (LSTM). LSTMs allow the usage of historical data in several steps in the future.

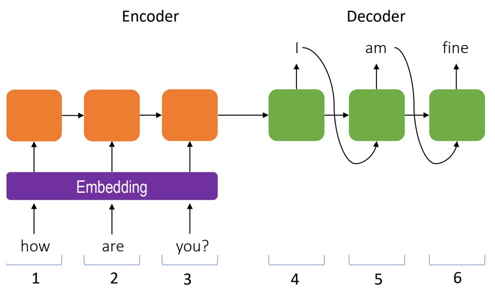


Figure: The general outline of sequence-to-sequence models consisting of an encoder and a decoder. The sequence *how are you?* is transformed to sequence *I am fine* in the figure.

Sequence-to-Sequence Based Virtualization (SSBV)

A sequence-to-sequence model is a special form of Recurrent Neural Network (RNNs), namely, Long Short Term Memory (LSTM). LSTMs allow the usage of historical data in several steps in the future.

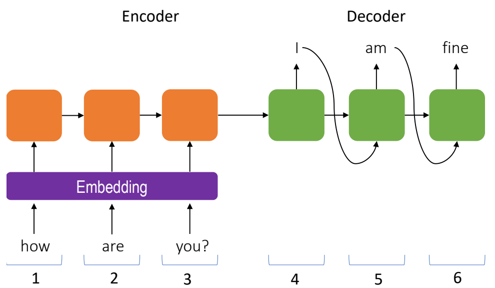


Figure: The general outline of sequence-to-sequence models consisting of an encoder and a decoder. The sequence *how are you?* is transformed to sequence *I am fine* in the figure.

Sequence-to-Sequence Based Virtualization (SSBV)

We start with creating a vocabulary with the letters and characters in interaction repository. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input.

Sequence-to-Sequence Based Virtualization (SSBV)

We start with creating a vocabulary with the letters and characters in interaction repository. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input.

The encoder learns to encode an input sequence into a vector and the decoder learns to decode this vector back to the output sequence.

Sequence-to-Sequence Based Virtualization (SSBV)

We start with creating a vocabulary with the letters and characters in interaction repository. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input.

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 our sequence-to-sequence model, we use prefixes of interaction traces, hence the model is learned iteratively.

Sequence-to-Sequence Based Virtualization (SSBV)

We start with creating a vocabulary with the letters and characters in interaction repository. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input.

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 our sequence-to-sequence model, we use prefixes of interaction traces, hence the model is learned iteratively.

Table: Sample inputs and the outputs used to train our sequence-to-sequence model.

Input	Output
createEvent event1 12/02/18	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

Table of Contents

1 Introduction

2 Related Work

3 Method

4 Evaluation

5 Conclusions

Evaluation

We used two services in evaluation, namely, a proprietary **Airline Ticketing Service** (ATS) and **Google Calendar API** (Calendar).

Evaluation

We used two services in evaluation, namely, a proprietary **Airline Ticketing Service** (ATS) and **Google Calendar API** (Calendar).

Correctness Evaluation

CBV:

- Exact Matching Ratio (EMR)
- Subset Matching Ratio (SMR)
- Micro averaged F-score (F_{micro})
- Macro averaged F-score (F_{macro})

SSBV:

- Accuracy

Evaluation

We used two services in evaluation, namely, a proprietary **Airline Ticketing Service** (ATS) and **Google Calendar API** (Calendar).

Correctness Evaluation

CBV:

- Exact Matching Ratio (EMR)
- Subset Matching Ratio (SMR)
- Micro averaged F-score (F_{micro})
- Macro averaged F-score (F_{macro})

SSBV:

- Accuracy

Performance Evaluation

Performance refers to the training time of the models for both of CBV and SSBV.

Evaluation

This is a multiclass-multilabel classification problem.

Exact Matching requires all the classes predicted for an input to be true. **Exact Matching Ratio** is number of predictions satisifying exact matching over number of all predictions.

On the other hand, **Subset Matching Ratio** is number of all correctly predicted classes over number of all classes predicted.

Micro- and macro-averaged F-scores are multiclass extensions of simple binary classification F-score.

Evaluation

Our experimental design:

- We collected 400 traces for each of the services and each trace contains 10 interactions (request-response pairs).

Evaluation

Our experimental design:

- We collected 400 traces for each of the services and each trace contains 10 interactions (request-response pairs).
- We compared CBV and SSBV with another tool in the literature, namely, EFSM Tool [8].

Evaluation

Our experimental design:

- We collected 400 traces for each of the services and each trace contains 10 interactions (request-response pairs).
- We compared CBV and SSBV with another tool in the literature, namely, EFSM Tool [8].
- For CBV method, we use Weka implementation of RIPPER and for SSBV method, we created models using Tensorflow library.

Evaluation

Our experimental design:

- We collected 400 traces for each of the services and each trace contains 10 interactions (request-response pairs).
- We compared CBV and SSBV with another tool in the literature, namely, EFSM Tool [8].
- For CBV method, we use Weka implementation of RIPPER and for SSBV method, we created models using Tensorflow library.
- We employed 5-fold cross validation for CBV and EFSM Tool.

Evaluation

Our experimental design:

- We collected 400 traces for each of the services and each trace contains 10 interactions (request-response pairs).
- We compared CBV and SSBV with another tool in the literature, namely, EFSM Tool [8].
- For CBV method, we use Weka implementation of RIPPER and for SSBV method, we created models using Tensorflow library.
- We employed 5-fold cross validation for CBV and EFSM Tool.
- Experiments were run on a server with 16 GB memory and Intel Xeon E5-2630L v2 2.40GHz CPU.

Evaluation

Table: Parameters selected in experiments.

Method	Parameters
CBV (RIPPER)	$minNo = 1$
SSBV (Tensorflow)	$hidden_size = 25$ $batch_size = 128$ $layers = 2$ $epochs = 1$ $iteration = 1000$
EFSM Tool (J48)	*default*

Our primary concern in choosing those parameters is maximizing the correctness of the models.

Evaluation

Table: Correctness results of CBV, SSBV and EFSM Tool for different values of k where k is the number of previous interactions considered.

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

Evaluation

Table: Correctness results of CBV, SSBV and EFSM Tool for different values of k where k is the number of previous interactions considered.

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

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.

Evaluation

Table: Performance results of CBV, SSBV and EFSM Tool for different values of k where k is the number of previous interactions considered. 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

Evaluation

Table: Performance results of CBV, SSBV and EFSM Tool for different values of k where k is the number of previous interactions considered. 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

If time is not in the first place, 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.

Table of Contents

1 Introduction

2 Related Work

3 Method

4 Evaluation

5 Conclusions

Conclusions

Service virtualization is getting popular with the rise of multi-layered and service oriented architectures. In this work, we developed techniques for automatically creating stateful virtual services.

Conclusions

Service virtualization is getting popular with the rise of multi-layered and service oriented architectures. In this work, we developed techniques for automatically creating stateful virtual services.

We presented machine learning based methods to create virtual services namely CBV and SSBV.

- CBV transforms response prediction problem into a classification problem.

Conclusions

Service virtualization is getting popular with the rise of multi-layered and service oriented architectures. In this work, we developed techniques for automatically creating stateful virtual services.

We presented machine learning based methods to create virtual services namely CBV and SSBV.

- CBV transforms response prediction problem into a classification problem.
- In SSBV, we employ sequence-to-sequence models for response generation.

Conclusions

Our evaluations demonstrate that the techniques introduced in this work are successful in terms of the defined metrics.

Conclusions

Our evaluations demonstrate that the techniques introduced in this work are successful in terms of the defined metrics.

In future, we plan to

- reduce training time of models while preserving correctness as high as SSBV's correctness for stateful service virtualization.

Conclusions

Our evaluations demonstrate that the techniques introduced in this work are successful in terms of the defined metrics.

In future, we plan to

- reduce training time of models while preserving correctness as high as SSBV's correctness for stateful service virtualization.
- infer the state machine of a service using recorded requests and responses.

QA

Thank you for listening. Questions?

Bibliography I

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*, pages 123–132. ACM, 2013.

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, 40(1):1–8, 2015.

H. Eniser and A. Sen.
Fancymock: Creating virtual services from transactions.
In Proceedings of the 33rd ACM SAC, 2018.

D. L. Giudice.
Service virtualization and testing solutions.
Forrester Wave, 2014.

Bibliography II

T. E. Murphy and N. Wilson.
Magic quadrant for integrated software quality suites.
Gartner Research, 2013.

J.-G. Schneider, P. Mandile, and S. Versteeg.
Generalized suffix tree based multiple sequence alignment for service virtualization.
In *Software Engineering Conference (ASWEC), 2015 24th Australasian*, pages 48–57. IEEE, 2015.

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*, pages 202–211. ACM, 2016.

Bibliography III

N. Walkinshaw, R. Taylor, and J. Derrick.
Inferring extended finite state machine models from software executions.
Empirical Software Engineering, 21(3):811–853, 2016.

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.

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 $content \in C_{req}$.

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 $content \in C_{res}$.

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 $content \in C_{res}$.
- An **interaction** is defined as a request response pair: (req, res) with $req \in Req$ and $res \in Res$.

Definitions

- 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), ..., (req_n, res_n).

Definitions

- 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.

Definitions

- 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.
- **History**, h, of a request req_i,
  $h_{req_i} = (req_1, res_1), \ldots, (req_{i-1}, res_{i-1}), (req_i)$,
  that is, the trace ends with req_i and the corresponding response is absent.

Definitions

- 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.
- **History**, h, of a request req_i,
  $h_{req_i} = (req_1, res_1), \ldots, (req_{i-1}, res_{i-1}), (req_i)$,
  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:
  $h^k_{req_i} = (req_{i-k}, res_{i-k}), (req_{i-k+1}, res_{i-k+1}), \ldots, (req_i)$.

Definitions

- *The set of all histories for all requests in all interaction traces* is denoted by $\mathbf{H}$.

Definitions

- *The set of all histories for all requests in all interaction traces* is denoted by $\mathbf{H}$.
- *The set of all k histories of all requests in all interaction traces* is denoted by $\mathbf{H_k}$.

Definitions

- *The set of all histories for all requests in all interaction traces* is denoted by $\mathbf{H}$.
- *The set of all k histories of all requests in all interaction traces* is denoted by $\mathbf{H_k}$.
- A **stateful service**, $StatefulS : H \rightarrow Res$, is a function from the set of histories to the set of responses.

Definitions

- *The set of all histories for all requests in all interaction traces* is denoted by $\mathbf{H}$.
- *The set of all k histories of all requests in all interaction traces* is denoted by $\mathbf{H_k}$.
- A **stateful service**, $StatefulS : H \rightarrow Res$, is a function from the set of histories to the set of responses.
- A **stateless service**, $StatelessS : Req \rightarrow Res$, is a function from the set of requests to the set of responses.

Definitions

- *The set of all histories for all requests in all interaction traces* is denoted by $\mathbf{H}$.
- *The set of all k histories of all requests in all interaction traces* is denoted by $\mathbf{H_k}$.
- A **stateful service**, $StatefulS : H \rightarrow Res$, is a function from the set of histories to the set of responses.
- A **stateless service**, $StatelessS : Req \rightarrow Res$, is a function from the set of requests to the set of responses.
- 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 synthesized responses, where k equals to 1 for a stateless service.

Metrics

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

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

where

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

E: expected outputs **P**: predicted outputs

n: the number of tests in validation phase

p: the number outputs predicted.

Metrics

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

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

where

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

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

Set $C = \{c_1, c_2, \ldots, c_n\}$ is the set of all classes to be predicted.