2018 IEEE International Conference on Web Services

Functionality-oriented Microservice Extraction Based on Execution Trace Clustering

Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui
Xi'an Jiaotong University
wx_jin@stu.xjtu.edu.cn,{tingliu,qhzheng}@mail.xjtu.edu.cn,cuidi@sei.xjtu.edu.cn

Yuanfang Cai
Drexel University
yfcai@cs.drexel.edu

***Abstract*—The main task of microservice extraction is to find which software entities (e.g., methods, classes) should be grouped together from existing monolithic software as candidate microservices, responsible for specific functionalities and evolving independently. Current methods extract microservices by analyzing source code and following the assumption that "classes with strong relation should be in the same service", which originates from software structure analysis. We find that 1) many program behaviors cannot be explicitly reflected in source code, and 2) the relation at code-level is not equivalent to the same functionality. Thus, we propose a functionality-oriented microservice extraction (FoME) method in this study by monitoring program dynamic behavior and clustering execution traces. Instead of source code analysis, the execution traces of a program are applied to group source code entities that are dedicated to the same functionality. We also construct a systematic measurement of microservice by integrating five complementary metrics of service cohesion and coupling. These metrics measure *Functional Independence* of microservices. That is, it qualifies whether a microservices can have its own responsibilities independently. In the experiment, our method is compared with three state-of-the-art methods on four open-source projects. The microservice candidates generated using our method present similar functional cohesion to the services produced using the other methods, but have considerably looser coupling measurements (dramatically reducing measurements of *IRN* and *OPN*).**

***Keywords*-microservice; executing trace; quality metrics;**

I. INTRODUCTION

Microservice-based systems aims to integrate a suite of smaller services into an application. Each service runs in its own process[1] and can evolve independently. These services are typically built around business capabilities and are responsible for their own functionalities. Researchers recommend that extracting microservices from existing monolithic code base is easier than constructing microservices from scratch [1]. This extraction process begins with a monolith code base, followed by classifying related functionalities into individual business capability groups, then implementing each group as a service [2]. In industrial practice, leading enterprises have been extracting microservices from monoliths, such as Netflix and Amazon.

The main task of microservice extraction is to recommend promising microservice candidates to software architects or developers, such that they do not have to manually conduct this labor-intensive work [3]. This extraction process is similar to existing software clustering or software decomposition methods, such as Bunch [4] and ACDC [5]. The problem is that, existing methods process source code and their syntactical relations to facilitate software clustering based on various rationales, such as coupling and cohesion [6], or naming convention [7]. However, no evidence proves that such relations definitely mean the same business capability. Therefore, we need further explore how entities with similar business logic can be found to construct a service.

To address this problem, we first leverage execution traces collected at runtime to guide microservice extraction. Execution traces are able to expose actual software behavior accurately [8]. A lot of tools can help obtain execution traces [9] [10]. Here we used Kieker [11] to monitor four software projects, in which 204 unique execution traces were extracted. This study shows that an execution trace inherently represent a business function. We also contribute an execution-oriented clustering method towards grouping similar functionalities as a service. We use **FoME** (**F**unctionality-**o**riented **M**icroservice **E**xtraction) to refer to the proposed method. Concretely, we cluster execution traces to recommend microservice skeletons, and then apply two strategies, namely, "Move class" and "Pull up class", to deal with crossovers overlapping between services.

Research on microservices remains in its early stage, and no systematic evaluation for microservice extraction is yet available [12][13]. Therefore, to estimate whether the proposed method can produce microservices with better functional independence, we integrate five metrics to measure *Functional Independence* quality of extracted microservices.

In summary, this paper makes the following contributions:

- We utilize runtime execution traces of software systems in microservice extraction. To our knowledge, this work is the first to leverage execution traces for facilitating microservice extraction.
- We design an execution trace-oriented clustering method to achieve functionality-based microservice extraction. This approach clusters source code entities that are dedicated to the same functionalities.
- We construct a systematic measurement by collecting five complementary metrics of service cohesion and coupling. This measurement system measures whether the extracted microservice candidated can present better

[1]https://martinfowler.com/articles/microservices.html

978-1-5386-7247-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICWS.2018.00034

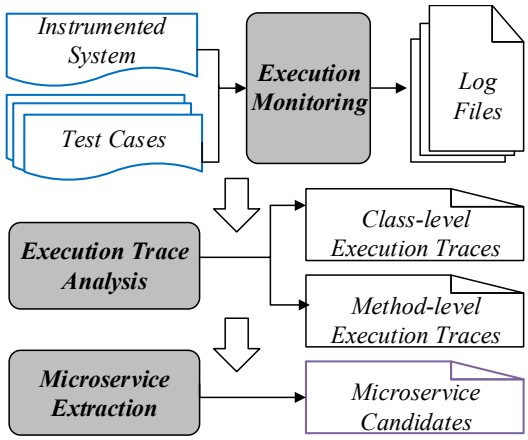


Figure 1. Overall process of the proposed method

Functional Independence.

The rest of this paper is summarized as follows. Section II describes our proposed approach for extracting microservices from an existing monolithic system. Section III illustrates the collected metrics for measuring *Functional Independence*. Section IV presents the experiments. Sections V and VI present threats to our work and related works. Section VII provides the conclusion.

II. Methodology

As shown in Figure 1, our method consists of three main steps. 1) Execution Monitoring. We apply Kieker 1.13 [11] to insert probes into the target system. The *Instrumented System* is executed using the pre-prepared *Test Cases* that cover as many functions as possible. The execution paths are recorded and stored in *Log Files*. 2) Execution Trace Analysis. From *Log Files*, we analyze execution traces to generate *Class-level* and *Method-level Execution Traces*. 3) Microservice Extraction. Algorithms are proposed to generate *Microservice Candidates* that consist of classes, interfaces and application programming interfaces (APIs). The first step is implemented by following the Kieker user guide. The details of Step 2 and 3 are explained in this paper.

A. Execution Trace Analysis

As shown in Figure 2(a), the execution of JPetstore[2] is monitored and recorded in *Log Files* with Kieker. Each line corresponds to one record, consisting of ten items: $Type$, $SeqID$, $Method$, $SessionID$, $TraceID$, Tin, $Tout$, $HostName$, Eoi and Ess. Five items are used to generate the execution traces. $Method$ denotes the complete signature of an invoked method, including modifier, class name, method name, and parameter list. $SessionID$ is a globally unique ID labeling a session. $TraceID$ is a globally unique ID labeling a trace. Eoi and Ess are the calling order and the depth of the calling stack of the method.

We define $LOG = \{rec\}$ is a set of records generated by a monitored system. For each record $rec \in LOG$, it includes five items mentioned above: *Method*, *SessionID*, *TraceID*, *Eoi* and *Ess*.

We define LOG_t as the completed execution log of a function t: $LOG_t = \{rec | rec \in LOG, and\ \forall rec_i, rec_j \in LOG_t, f_{se}(rec_i) = f_{se}(rec_j),\ f_{tr}(rec_i) = f_{tr}(rec_j)\}$. $f_{se} : rec \rightarrow SessionID$, $f_{tr} : rec \rightarrow TraceID$, $f_{eoi} : rec \rightarrow Eoi$, and $f_{ess} : rec \rightarrow Ess$. All records rec in LOG_t present the same *SessionID* and *TraceID*, corresponding to all executions about a specific function. This is inherent in logs, since the used instrumentation technique can end-to-end (e2e) track executions from starting to finishing a function through different code layers. Even though a function execution may involve more than one thread, or more than one process, its recorded recs will always contain the same *SessionID* and *TraceID*.

In terms of (*SessionID, TraceID*), we can form a partition ζ of LOG: $\zeta = \{LOG_t\}$ and $\forall LOG_t, LOG_k \in \zeta, t \neq k, LOG_t \neq \emptyset, LOG_k \neq \emptyset, LOG_t \cap LOG_k = \emptyset$. That is, $LOG = \bigcup_{LOG_t \in \zeta} LOG_t$.

We then try to extract method calls from $LOG_t \in \zeta$. Define that (m_i, m_j) (i.e. $m_i \rightarrow m_j$) is a relation of *Method Call*, in which m_i is *caller* and m_j is *callee*. rec_i is the record including m_i and rec_j is the record including m_j. We identify a *Method Call* relation based on the rule:

If $f_{ess}(rec_i) = f_{ess}(rec_j) - 1$, $f_{eoi}(rec_i) < f_{eoi}(rec_j)$, and $\forall rec_k \in S = \{rec_k | f_{ess}(rec_k) = f_{ess}(rec_i), k \neq i\}, f_{eoi}(rec_k) < f_{eoi}(rec_i)$, we conclude m_i is *caller* of m_j.

For a LOG_t, we define its corresponding method-level execution trace as an ordered sequence $tr : \langle (m_{i1}, m_{j1}),\ (m_{i2}, m_{j2}),\ ..., (m_{in}, m_{jn}) \rangle$, where $f_{eoi}(rec_{j1}) < f_{eoi}(rec_{j2}) < ... < f_{eoi}(rec_{jn})$. A method-level execution trace presents the method-calling relation and invocation order. Then processing all $LOG_t \in \zeta$ in above way, we can finally get: $TR = \{tr\}$ and $|TR| = |\zeta|$. TR is called *Execution Traces*, or more specifically, *Method-level Execution Traces*.

Removing duplicated elements from TR, we finally get a new TR, in which every element is a unique method-level execution trace which corresponds to one specific function.

For $tr_i \in TR$, we can extract the corresponding class-level execution trace d_ctrace_i. d_ctrace_i is an ordered sequence $d\_ctrace_i : \langle (c_{i1}, c_{j1}),\ (c_{i2}, c_{j2}),\ ..., (c_{in}, c_{jn}) \rangle$, where c_i denotes the class which contains method m_i. A class-level execution trace is a class-granularity calling sequence, showing which classes are dedicated to the same functional execution. Then for TR, we finally get: $D\_CTrace = \{d\_ctrace_i\}$. D_CTrace is called *Class-level Execution Traces*. Considering the example of Figure 2(a), its class-level and method-level execution traces are

[2]https://github.com/mybatis/jpetstore-6

Algorithm 1: Execution trace clustering

Input: $D\_CTrace = \{d\_ctrace_i\}$,
$d\_ctrace_i = \{c_{i1}, c_{i2}, ..., c_{in}\}$,
$D\_TCDep = \{d\_tcdep_{(d\_ctrace_i, c_j)}\}$,
$M \in [1, size(D\_CTrace)]$

Output: $cluster_k = \{d\_ctrace_i\}$,
$cluster_k \subset D\_CTrace$, $1 \leq k \leq M$

```
for (j = 0; j < size(D_CTrace); j++) do
    cluster_j = d_ctrace_j;
end
i = size(D_CTrace);
repeat
    for (j = 0; j < i; j++) do
        for (k = 0; k < i; k++) do
            simMatrix[j][k] =
              Sim_Func(cluster_j, cluster_k, D_TCDep);
        end
    end
    (j, k) = simMatrix.index(max(simMatrix));
    cluster_j.add(cluster_k); delete cluster_k;
    i = i - 1;
until i < M;
```

visualized in Figure 2(b) and 2(c).

B. Microservice Extraction: FoME

This section presents our approach FoME for extracting microservice candidates, i.e., their classes, interfaces, and APIs, from the aforementioned execution traces. The approach consists of three steps: 1) class clustering, 2) shared class processing, and 3) microservice candidate generation.

*1) **Class clustering**:* Microservice design should follow the "Single Responsibility Principle"[3]. This principle recommends that one service should be changed with one unique reason. A class-level trace corresponds to the execution of a function point of the original monolithic software. Therefore, we can group the classes dedicated to the same business logic as the skeleton of one microservice by clustering the execution traces.

We design an execution trace clustering algorithm (Algorithm 1). We use classic hierarchical clustering with a given parameter M. M is the number of clusters. $d\_tcdep_{(d\_ctrace_i, c_j)}$ is measured as the times that class c_j appears in d_ctrace_i. D_TCDep is a set of all $d\_tcdep_{(d\_ctrace_i, c_j)}$. $Sim\_Func(cluster_j, cluster_k, D\_TCDep)$ computes the similarity between $cluster_j$ and $cluster_k$ by utilizing the Jaccard Coefficient [14]. After clustering, the generated M clusters correspond to the skeletons of M services.

[3]https://www.infoq.com/articles/microservices-intro

*2) **Crossover processing**:* Similar to traditional software clustering methods, several classes are clustered into more than one service, called shared classes or crossovers. These classes are crossovers in class granularity among bounded contexts in terms of domain concept or logic [15].

In traditional software clustering, a crossover is assigned to a particular cluster if its dependency on that cluster is stronger than that on other clusters [16]. However, this approach is not the only viable one in practice. Some shared classes can also be extracted to a separate service. For example, when several services share classes, which are not dedicated to the business logic, these shared classes do not exhibit a strong functional relation to their located service skeletons. Accordingly, packaging the classes into a newly created service is appropriate. In this manner, the shared new service can be maintained separately. As long as the provided interfaces are not changed, frequent updates to the new service will not influence the services that use it.

In our work, two strategies, namely, "Move class" and "Pull up class", which are similar with the "Move method" and "Pull up method" in software code-level refactoring [17], are designed to further handle the aforementioned classes. Assume there is a shared class c_k which is clustered into two services' skeletons $cluster_i$ and $cluster_j$. We select a strategy by considering the dependency between the shared class and clusters. $D\_SCdep(cluster_i, c_k)$ computes the dependency between $cluster_i$ and c_k as follows:

$$D\_SCdep(cluster_i, c_k) = \frac{\sum_{j=1}^{size(cluster_i)} d\_tcdep(d\_ctrace_j, c_k)}{size(cluster_i)}, \quad (1)$$
$$d\_ctrace_j \in cluster_i, 1 \leq j \leq size(cluster_i),$$
$$d\_tcdep(d\_ctrace_j, c_k) \in D\_TCDep$$

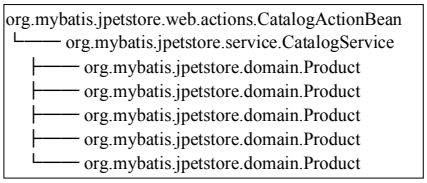
If a class, c_k, exhibits high dependency with one service (larger than a threshold), we can simply move this class into that service by using "Move class". If the dependency of class c_k is weak with $cluster_i$ and $cluster_j$, "Pull up class" is recommended; that is, the shared class c_k will be created as a new service $cluster_l$. "Pull up class" is suitable for shared classes that are relevant to tool sharing.

*3) **Microservice candidate generation**:* Integrating the aforementioned clustered classes with detailed *Method-level Execution Traces* TR, we can identify *Interfaces* and corresponding fine-grained *APIs* for each extracted microservice candidate. Here, *APIs* are fine-grained operations in the method level that correspond to inter-service calling methods. *Interfaces* correspond to the original classes of the monolith involved in the *APIs*. The *APIs* and *Interfaces* are provided by one extracted service to other extracted services.

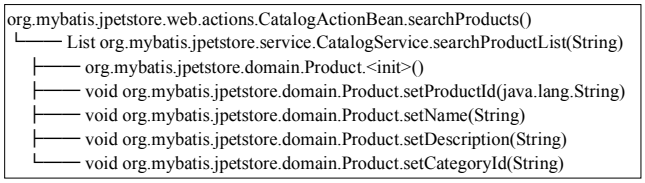
III. Evaluation Metrics

We aim at extracting microservices in a functionality-oriented way, so the produced microservice is expected

(a) Snippet of *Log Files*



(b) *Class-level execution trace*



(c) *Method-level execution trace*

Figure 2. Execution logs and traces of JPetstore

to present functional independence. That is, ideally, a microservice should present its own single responsibilities, such that evolving independently. More specifically, microservice should be functionally cohesive and decoupled from other services [1]. A functionally cohesive service classifies related functional behaviors into a group, and excludes unrelated ones. Athanasopoulos el al. Many cohesion metrics have been proposed such as [18] and [19]. [20] used cohesion to drive decomposition of service interface. When services are loosely coupled, a functional update to one service will not require a change to another service. Ouni et al. [21] used coupling and other metrics to detect antipatterns in web services. Adjoyan et al. [22] minimized the number of service interfaces to facilitate service identification. In general, high cohesion and loose coupling are complementary, and they have been investigated simultaneously in traditional software design research. However, research on microservice architecture remains in an early stage. No widely accepted evaluation metrics for microservice works is yet available [12][13].

In this work, we define a quality criteria named *Functional Independence*, qualifying the extent of independence to which microservices present their own functionalities. To quantify this attribute, from existing works we collect five metrics CHD (CoHesion at Domain level), CHM (CoHesion at Message level), IFN (InterFace Number), OPN (OPeration Number)and IRN (InteRaction Number), to build a systematic evaluation of *Functional Independence* for microservices. More concretely, CHD and CHM measure the functional cohesion of microservices; IFN, OPN and IRN measure the coupling between microservices. We will first define several parameters to explain these metrics. Then we will formally present the metrics.

S_C_i is a set of classes contained in $Service_i$. S_I_i is a set of interfaces provided by $Service_i$. Op_i is an API provided by interface I_i. Op_i includes its name ($name_i$), returning parameters (res_i), and input parameters (pas_i). T_{Op_i} is used to describe the set of domain terms extracted from the $name_i$ of Op_i.

$Services = \bigcup_{i=1,2,...,K} Service_i, Service_i = (S\_C_i, S\_I_i)$

$S\_C_i = \{c_{i1}, c_{i2}, ..., c_{ik_i}\}, S\_I_i = \{I_{i1}, I_{i2}, ..., I_{in_i}\}$

$I_i = \{Op_{i1}, Op_{i2}, ..., Op_{il_i}\}, Op_i = (res_i, name_i, pas_i)$

$pas_i = \{p_{i1}, p_{i2}, ..., p_{im_i}\}, res_i = \{r_{i1}, r_{i2}, ..., r_{ip_i}\}$

Metric 1: *CHM*. CHM is used to measure the average cohesion of service interfaces at message level. It is a variation of Lack of Message-Level Cohesion LoC_{msg}[20]. $CHM = 1 - LoC_{msg}$. CHM is more, the better. It's formalized as follows:

$$CHM = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHM_j}{\sum_{i=1}^{K} n_i} \tag{2}$$

Where,

$$CHM_j = \begin{cases} \frac{\sum_{(k,m)} f_{simM}(Op_k, Op_m)}{|I_i| \times (|I_i|-1)/2}, & if |I_i| \neq 1 \\ 1, & if |I_i| = 1 \end{cases}$$

$f_{simM}(Op_k, Op_m) = (\frac{|res_k \cap res_m|}{|res_k \cup res_m|} + \frac{|pas_k \cap pas_m|}{|pas_k \cup pas_m|})/2.$

n_i is the number of provided interfaces of Microservice i. K is the number of microservices identified from a monolithic software. CHM_j measure cohesion of I_i at message level. Op_k and Op_m are operations provided by I_i, and $k \neq m$. f_{simM} computes the similarity between two operations at message level. It is the average of similarity in terms of input message (eg. parameters) similarity and output (eg. returnings).

Metric 2: *CHD*. CHD is used to measure the average

cohesion of service interfaces at domain level. It is a variation of Lack of Domain-Level Cohesion LoC_{dom}[20]. $CHD = 1 - LoC_{dom}$. CHD is more, the better. It's formalized as follows:

$$CHD = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHD_j}{\sum_{i=1}^{K} n_i} \tag{3}$$

Where,

$$CHD_i = \begin{cases} \frac{\sum\limits_{(k,m)} f_{simD}(Op_k, Op_m)}{|I_i| \times (|I_i|-1)/2}, & if |I_i| \neq 1 \\ 1, & if |I_i| = 1 \end{cases}$$

$f_{simD}(Op_k, Op_m) = |T_{Op_k} \cap T_{Op_m}| / |T_{Op_k} \cup T_{Op_m}|$

n_i and K are same symbols with those defined in ***CHM*** formulation. f_{simD} computes the similarity between two operations at domain level. TP_i is used to describe the set of domain terms extracted from the $Name_i$ of Op_i.

Metric 3: *IFN*. IFN indicates the number of interfaces [22] provided by an extracted service to other services averagely. IFN is smaller, the better.

$$IFN = \frac{1}{K} | \bigcup_{i=1,2,\ldots,K} Service\_I_i | \tag{4}$$

Metric 4: *OPN*. OPN denotes the number of operations provided by the extracted microservices. OPN is less, the better.

$$OPN = | \bigcup_{i=1,2\ldots,\sum_{i=1}^{i=K} n_i} I_i | \tag{5}$$

Metric 5: *IRN*. IRN represents the number of method calls across two services. IRN is smaller, the better [1].

$$IRN = \sum_{(Op_j, Op_k)} w_{j,k} \tag{6}$$

Where, $w_{j,k}$ is the calling frequency from Op_j to Op_k. Op_j and Op_k are involved in inter-service interactions between the extracted microservices.

IV. EXPERIMENTS

We first introduce the experimental setup, then use the case of JPetstore to illustrate the result. Finally, we evaluate our method by comparing it with three existing methods, using the metrics presented in Section III.

A. Setup

1) ***Subjects****:* We study monolithic web applications because their server-side codes are typically packaged into a single unit, such as WAR or EAR files, in the earliest days. Large-scale monolithic web applications usually suffer from maintainability and scalability issues due to their size and complexity. Our experiments aim to extract microservice candidates for the server-side of web applications.

We collect four web applications as subjects, and the basic information of these applications is provided in Table I.

Table I
SUBJECT PROJECTS INTRODUCTION

Subject	Version	Description	SLOC	Class	Test case
JPetstore	6.0.2	e-store	1438	24	15
Springblog	2.8.0	blogging	3583	85	33
JForum	2.1.9	forum	29550	340	69
Roller	5.2.0	blogging	47602	534	87

These monolithic web systems differ in scale and are all implemented using JAVA. JPetstore is a pet store e-commerce demo of online website. JForum is a discussion board system. Springblog is a blogging system. Roller is a full-featured, multi-user and group-blog server suitable for blog sites. Column *SLOC* shows the lines of JAVA code. Column *Class* counts classes without inner classes. These subject projects have been collected in *https://github.com/wj86*.

2) ***Test cases****:* We designed test cases as input to drive software execution. The test cases contained in the original repository are not designed to generate execution traces that accurately indicate business functionality. Instead, many of these test cases are unit tests and use mock approaches. Thus, new test cases should be designed. The explicit function entry of targeted software systems is GUI. Accordingly, we designed test cases by utilizing all GUI entries to cover the maximum number of software system functions. In particular, column *Test case* of Table I shows the number of unique test cases we have designed. The detailed test cases can be found in *https://github.com/wj86/FoME*.

B. Case Study on JPetstore

We use JPetstore as an example to illustrate the processing results of our method. Figure 2(a) shows the *Logging Files* after executing one test case. The class-level and method-level execution trace of this test case are visualized in Figure 2(b) and 2(c) respectively. Similarly, all 15 method-level and 15 class-level execution traces can be generated when executing the designed 15 test cases.

Figure 3 illustrates the component model of the target microservices that contains three microservices that must be extracted. The three microservices are *Service_Account*, *Service_Product*, and *Service_Order*. *Service_Order* requires three interfaces provided by other two services to fulfill its functionality. Figure 4 depicts how classes are grouped into three microservice candidates. We identify the interfaces and fine-grained API signatures (Figure 5) provided by the microservices using 15 method-level execution traces. Our proposed method can generate the three candidates automatically and produce the aforementioned information to assist architects or developers, reducing the manual effort needed to extract microservice candidates.

C. Experimental Evaluation

1) ***Compared methods****:* Our method is compared with the following state-of-the-art methods: **LIMBO** [23], **WCA**

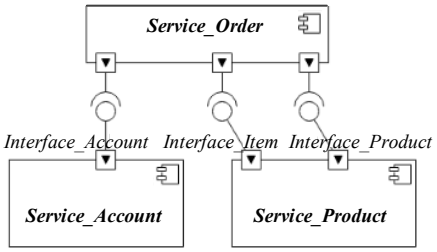


Figure 3. Three microservices extracted from JPetstore

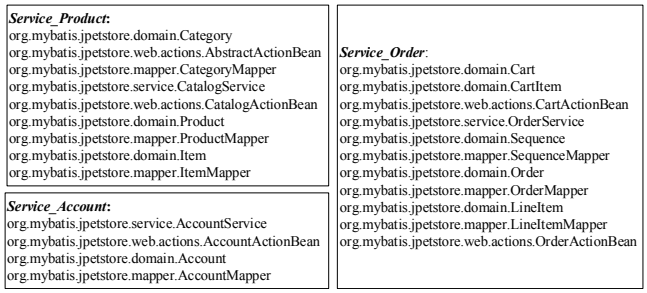


Figure 4. Classes in each microservice

[14], and **MEM** (Microservice Extraction Model) [3]. **WCA** is a hierarchical clustering method that leverages two measures to determine the similarity between classes: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). Here we use WCA with UENM for comparison because it performs better than UE [24]. **LIMBO** uses information loss to measure the distance between classes. **MEM** cuts the class graph of the original monolith. Relation on edges is extracted using three strategies, namely, logical, semantic, and contributor couplings. In this paper, we use semantic coupling-based approach for comparison because the other two strategies cover an extremely small number of classes in our target projects.

*2) **Coverage analysis and the baseline**:* Coverage refers to the proportion of classes that are covered in the execution traces in our method, or the classes that participated in the

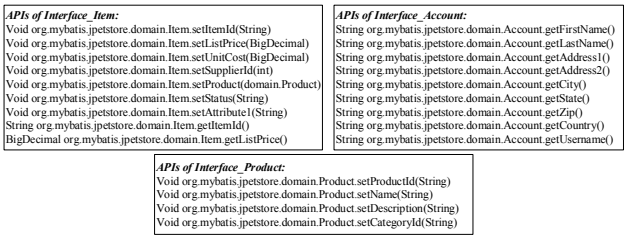


Figure 5. Interfaces and APIs provided by each microservice

Table II
CLASS COVERAGE OF DIFFERENT METHODS

Subject	LIMBO	WCA	MEM	FoME
JPetstore	100%	100%	95.83%	100%
Springblog	91.76%	91.76%	85.88%	72.94%
JForum	97.15%	97.15%	60.76%	61.39%
Roller	96.22%	96.22%	77.94%	77.31%

Table III
UNCOVERED CLASSES ANALYSIS OF SPRINGBLOG

Type	Proportion	Description
NoBehavior	7.06%	Classes only having member variables but no member methods.
Exception	2.35%	Classes responsible for exceptions.
3rdPartyService	9.41%	Classes accessing third party services.
Other	8.24%	Other reasons.

clustering using the above existing methods. Table II shows the coverage of all the four methods. We can observe that the coverage is different in various methods and is not 100% in most cases. The coverage of LIMBO and WCA is always the same, because both methods are based on class structural dependency from source codes. MEM is based on class-to-class semantic coupling through information retrieval techniques. Each pair of classes can be assigned a value within [0.0, 1.0], which indicates the probability that they are semantically related. Here, we empirically select 0.7 as the threshold to determine the relation.

The further investigation on the coverage of FoME shows that uncovered classes could be categorized as four types as shown in Table III, in which Springblog is used as an example: 1) *NoBehavior* classes. They are in `models.dto.*`. Execution traces cannot reflect these classes without behaviors. 2) *Exception* classes. They are included in `error.*`. Errors will not happen because the designed test cases are executed successfully. 3) *3rdPartyService* classes. Located in `support.web.*`, these classes are mainly for accessing Markdown, YouTube linking, and syntax highlighting. 4) *Other* classes. We can see those classes are excluded due to their irrelevance to business logic.

To conduct a fair comparison, we use the same set of classes covered by executing traces as the baseline, because of the accuracy of dynamic executing traces. We obtain WCA, LIMBO, and MEM clusterings as follow: 1) WCA and LIMBO. The two methods utilize software structural dependency as input. The direct application of these methods on baseline classes may produce clusterings that do not cover all the baseline classes. Therefore, we first run these methods using all classes, and then filter out the results using the baseline classes. 2) MEM. Semantic similarity can be measured for most class-class pairs. Therefore, we directly apply this method on the baseline classes.

Table IV
MEASUREMENT RESULTS

Subject (Microservice)	Metric	Methods LIMBO	WCA	MEM	FoME
JPetstore(**3**)	*CHD*	0.6-0.7	0.6-0.7	0.6-0.7	0.6-0.7
	CHM	0.5-0.6	0.5-0.6	0.5-0.6	0.7-0.8
	IFN	2.3333	1.0000	0.6667	1.0000
	OPN	68	44	39	22
	IRN	329	68	48	35
	Total	1	1	2	4 *
Springblog(**7**)	*CHD*	0.7-0.8	0.8-0.9	0.8-0.9	0.8-0.9
	CHM	0.6-0.7	0.7-0.8	0.7-0.8	0.7-0.8
	IFN	5.2857	2.0000	1.4286	1.0000
	OPN	147	33	21	7
	IRN	238	52	30	26
	Total	0	2	2	5 *
JForum(**9**)	*CHD*	0.6-0.7	0.5-0.6	0.6-0.7	0.7-0.8
	CHM	0.6-0.7	0.5-0.6	0.6-0.7	0.7-0.8
	IFN	3.5556	6.4444	4.2222	2.5556
	OPN	94	25	11	70
	IRN	993	691	145	97
	Total	0	0	1	4 *
Roller(**16**)	*CHD*	0.7-0.8	0.8-0.9	0.8-0.9	0.8-0.9
	CHM	0.6-0.7	0.6-0.7	0.6-0.7	0.6-0.7
	IFN	13.3125	1.1250	1.4375	1.7500
	OPN	1062	56	66	56
	IRN	46964	4966	2786	1441
	Total	1	4 *	2	4 *

[a] Microservice is the number of extracted microservices.
[b] "Total" shows the number of metrics, in which a method outperforms others.
[c] * highlights the best one according to 'Total'.

*3) **Evaluation results**:* We configure all methods to extract the same number of target microservices. The number of microservices is determined by the domain experts. In Table IV, the first column shows the project name and the number of services to be extracted. Row *CHM* and *CHD* present cohesion measures, the higher the better. Since small differences between *CHM* (or *CHD*) scores are insignificant, we use a scope (e.g., [0.6, 0.7]) of a cohesion score instead of a specific value (e.g., 0.655). The measurements including *IFN*, *OPN* and *IRN*, the lower the better.

The results indicate that FoME outperforms other methods in most metrics. Specifically, nearly all of the methods can achieve functionally cohesive microservices, whereas our method generates significantly loosely coupled microservices. In particular, when compared with the secondary best measures, our method can reduce coupling dramatically in metrics of *OPN* and *IRN*. In general, the proposed method can obtain microservices with better *Functional Independence* over existing methods.

V. THREATS TO VALIDITY

Even though execution trace is a promising basis for microservice extraction, it suffers from several limitations. First, all execution traces are generated using given test cases and captured using instrumentation. Consequently, the quality of test cases becomes critical: these test cases should cover the maximum number of business functionalities. Otherwise, function-relevant classes may not be executed and will not be grouped into suitable services. This limitation is acceptable because the architects in charge of microservice extraction should have sufficient knowledge of target systems, including test cases, execution environment, parameter setting. In addition, state-of-the-art instrumentation techniques will result in additional overhead in system execution and may even cause system crash down. Our method provides a post-mortem analysis under a testing environment. It will not influence the runtime system in a production environment.

Second, microservice architecture is still in its infancy. Estimation is lacking in nearly all existing works [12][13]. As a consequence, we cannot evaluate the proposed method against a widely-accepted measuring system. To address this problem, we evaluated the proposed method from two dimensions. First, we have collected cohesion and coupling scores to systematically evaluate extracted microservices from the perspective of maintainability. Second, we have made comparions with three state-of-the-art methods: WCA, LIMBO, and MEM. WCA and LIMBO works are traditional software decomposition methods, which also have been selected as comparing methods in existing works [24][16]. MEM is the latest microservice extraction method.

Finally, our work aims to provide assistance, and not complete substitution to manual microservice extraction, which is a labor- and human wisdom-intensive work. The proposed method can help mitigate the burden of architects or developers, but there is still a long way to go before fully automated microservice extraction can be achieved, similar to auto-code generation, auto-debugging and auto-repair.

VI. RELATED WORK

Software Decomposition. Software decomposition aims to split a large system into smaller and manageable clusters. It is a general topic. WCA and LIMBO are classic decomposition methods that use static structural relation extracted from source code as input[14][24]. There are similar work dedicated to software architecture recovery [7]. Recently, [16] [24] performed a comparative analysis of six state-of-the-art techniques of software architecture recovery.

Microservice Extraction. Microservice extraction is a form of software decomposition in the realm of service-oriented architecture, a counterpart of traditional software decomposition. [25] conducted microservice extraction by starting with splitting a database. This approach requires manually splitting database tables into groups, which is also a tedious work. Learning from traditional software decomposition, [3] partitioned the class graph of a monolithic software into components similarly based on class-to-class coupling. They considered individual components as microservice candidates.

VII. CONCLUSION

In this paper, we propose a functionality-oriented microservice extraction method to automatically generate microservice candidates from a monolithic software system. We use execution traces to guide the grouping of source code entities that are dedicated to the same functionality. Compared with other methods using static source code, execution traces can better reflect business functionality. By measuring the *Functional Independence* quality criteria, the experiments using four open-source projects show that our method can extract microservice candidates with similar cohesion scores, but perform considerably better in terms of coupling scores than LIMBO, WCA and MEM.

ACKNOWLEDGMENT

This work was supported by National Key RD Program of China (2016YFB1000903), National Natural Science Foundation of China (61772408, U1766215, U1736205, 61721002, 61472318, 61532015, 61632015), Fok Ying-Tong Education Foundation (151067), Ministry of Education Innovation Research Team (IRT_17R86) and Project of China Knowledge Centre for Engineering Science and Technology.

REFERENCES

[1] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[3] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 524–531.

[4] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 50–59.

[5] V. Tzerpos and R. C. Holt, "Accd: an algorithm for comprehension-driven clustering," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE, 2000, pp. 258–267.

[6] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 24, 2016.

[7] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 552–555.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software:Evolution and Process*, vol. 25, pp. 53–95, 2013.

[9] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, 2018.

[10] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[11] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012, pp. 247–248.

[12] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 44–51.

[13] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 21–30.

[14] M. Chatterjee, S. K. Das, and D. Turgut, "Wca: A weighted clustering algorithm for mobile ad hoc networks," *Cluster computing*, vol. 5, no. 2, pp. 193–204, 2002.

[15] S. Millett, *Patterns, Principles and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[16] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, 2017.

[17] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[18] W. Jin, T. Liu, Y. Qu, Q. Zheng, D. Cui, and J. Chi, "Dynamic structure measurement for distributed software," *Software Quality Journal*, pp. 1–27, 2017.

[19] W. Jin, T. Liu, Y. Qu, J. Chi, D. Cui, and Q. Zheng, "Dynamic cohesion measurement for distributed system," in *International Workshop on Specification, Comprehension, Testing and Debugging of Concurrent Programs*. ACM, 2016, pp. 20–26.

[20] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2015.

[21] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 603–617, 2017.

[22] S. Adjoyan, A.-D. Seriai, and A. Shatnawi, "Service identification based on quality metrics object-oriented legacy system migration towards soa," in *SEKE: Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2014, pp. 1–6.

[23] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[24] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2013, pp. 486–496.

[25] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *CoRR*, vol. abs/1605.03175, 2016.