2018 25th Asia-Pacific Software Engineering Conference (APSEC)

Extracting Candidates of Microservices from Monolithic Application Code

Manabu Kamimura, Keisuke Yano, Tomomi Hatano, Akihiko Matsuo
Software Laboratories
Fujitsu Laboratories Ltd.
Kawasaki, Japan
Email: {kamimura.manabu, yano, hatano.tomomi, a_matsuo}@jp.fujitsu.com

***Abstract*—Technology that facilitates rapid modification of existing business applications is necessary and it has been reported that making the system more adaptable to change is the strongest driver for legacy system modernization. There has been considerable interest in service-oriented architectures or microservices which enables the system to be quickly changed. Refactoring and, in particular, re-modularization operations can be performed to repair the design of a software system. Various approaches have been proposed to support developers during the re-modularization of a software system. The common problem in these efforts is to identify from monolithic applications the *candidates of microservices*, i.e., the programs or data that can be turned into cohesive, standalone services; this is a tedious manual effort that requires analyzing many dimensions of software architecture views and often heavily relies on the experience and expertise of the expert performing the extraction. To solve this problem, we developed a method that identifies the candidates of microservices from the source code by using software clustering algorithm SArF with the relation of "program groups" and "data" which we defined. Our method also visualizes the extracted candidates to show the relationship between extracted candidates and the whole structure. The candidates and visualization help the developers to capture the overview of the whole system and facilitated a dialogue with customers. We report two case studies to evaluate our results in which we applied our method to an open source application and an industrial application with our results reviewed by developers.**

Keywords*—microservice architecture, software clustering, partition, visualization, program comprehension, modernization

I. Introduction

In the digital transformation of business using information communication and technology (ICT), business is rapidly changing and the technology that facilitates rapid modification of existing business applications is required. It has been reported that making the system more adaptable to change is the strongest driver for legacy system modernization [1]. The structure of existing business applications is often complex and changes have a far-ranging impact, which requires time consuming revisions [2], [3]. To resolve this problem, there has been considerable interest in a service-oriented architectures [4] or *microservices* [5], [6] to design applications in which business processes can be quickly changed. Microservice architecture structures an application as a set of loosely coupled services. These services are independently deployable which enables the system to be quickly changed.

The existing business application which is often referred as *monolithic application* [5], [6] as it is built on a single unit usually consists many business capabilities, which are the core functions to run the business. The existing monolithic application may be in large scale or complicated and microservices are recommended in such cases when it is hardly manageable. Serious efforts have been undertaken to move from monolithic architectures to microservice architectures. The common problem in these efforts is to identify from monolithic applications the *candidates of microservices* which includes the programs (such as program files and class files) or data (such as database tables, files, and data objects) that can be turned into cohesive, standalone services; this is a tedious manual effort that requires analyzing many dimensions of software architecture views and often heavily relies on the experience and expertise of the expert performing the extraction [6], [7]. For example, in defining microservices in previous works, defining building blocks and its relation is required which is a manual effort to start the process [8]. We aim to tackle the abovementioned extraction problem.

In this paper, we aim to identify the *candidates of microservices,* which include the programs and data from monolithic applications without relying on manual efforts for the initial step. Our method analyzes the source code with information as "entry points", which is the user's interaction points defined as the application programming interfaces (APIs), batch execution points or front-end screens, and visualizes the whole structure using city metaphor and produces the list of programs and data as the candidates of microservices and the "common data" which is accessed from many candidates of microservices. Our work is summarized as follows:

1. We propose a method which produces the candidates of microservices as the list of programs and data from the source code of monolithic applications.
   I. We use SArF software clustering algorithm [9] with data access [10] and the relationship between "program groups" and "data" which we defined as the input. Our work also visualizes the extracted candidates to show the relationship between extracted candidates and the whole structure [11].
2. We conducted two case studies.

I. We applied our method to an open source application of monolithic version and compared our results with those of the microservice version of the same application.

II. We also applied our method to an industrial application and our results were reviewed by the developers.

The remainder of this paper is organized as follows: Section II shows our problem. Section III explains our approach of extraction and visualization of candidates and Section IV and V show case studies and the preliminary results from applying our method to industrial systems. Related works are discussed in section VI and finally, the paper is summarized in Section VII.

II. Problem Definition

As mentioned in Section I, making the system more adaptable to change is the strongest driver for legacy system modernization because of the complex structure of the existing application. We analyzed the structure of the existing application because the new system replacing the existing system should have the same functions implemented in the existing application. However, the existing application is usually in a large scale and contains many related functions; therefore, the application needs to be decomposed.

We tried to decompose the existing application by a software clustering method. Software clustering is a technique which decomposes a given system into groups of software entities (e.g. programs, data) with manageable sizes [9]. Such decomposition can be used as the architectural knowledge and the high-level abstraction views of the system [2]. First we assigned programs (such as program files and class files) and data (such as database tables, files, and data objects) as nodes, and program calls (or function calls, method invocations) and data accesses (references and updates from the program) as dependencies (Figure 1). In the experiment, we used an SArF [9] algorithm with data access [10] which produced the clusters shown on the right of Figure 1. PG2 is only called by PG1 so the relationship between PG2 and PG1 is relatively strong. However, PG4 is called by both PG2 and PG3, so the relationship between PG2 and PG3 is relatively weak.

The results show that the logic and data closely related are extracted as clusters though the dependencies such as program calls remain. When PG1 is executed, it will call PG2 and PG4. PG4 is in an another cluster so both clusters are still related. When we target microservices, each service should be evolutional. The key property of microservices is the notion of independent replacement and upgradeability[4], [5], [6]. As we see in Figure 1, the clustering decomposes into two clusters but dependencies exist because both clusters depend on PG4, so that it cannot be separated when executed. Thus, when defining the candidates of microservices, dependencies should be avoided but the case of Figure 1 is not decomposable because dependent programs exist in other clusters.

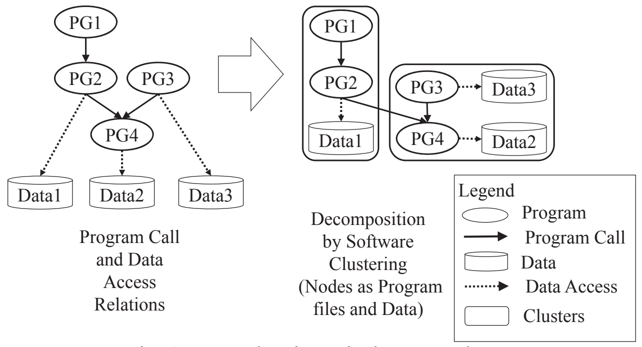
PG1
PG2
PG3
PG4
Data1
Data2
Data3
Program Call and Data Access Relations
Decomposition by Software Clustering (Nodes as Program files and Data)
Legend
Program
Program Call
Data
Data Access
Clusters

Fig. 1. Dependencies exist between clusters

III. Our Approach

A. Overview

To solve this problem, we propose a method which produces the candidates of microservices as a list of program and data from the source code of monolithic application. We extract the dependencies to other candidates as data access only because programs can be included into each candidate. Our method collects all the dependent programs related to each entry point which is the interaction point between the system and the user. We define this list as "program groups". We analyze the relationship between program groups and data to capture "the candidates of microservices" using SArF software clustering algorithm [9] with data access [10]. We classify data into common data which are updated or accessed from many candidates of microservices and local data included in the candidates of microservices. We visualize the results and this enables the whole structure to be captured [11]. The process is described in detail.

B. Extracting dependencies of program groups and data

We focus on avoiding dependencies between the candidates of microservices. We classified the dependencies into two groups that are between programs and between programs and data. Programs can be deployed into many execution modules such as hardcoded libraries, so that all programs which are necessary, such as called programs, can be included into one microservices to avoid dependencies with other services. Though the data have status information which is shared by many processes. Duplicating the data, such as database tables, is expensive to support because the data consistency becomes the problem. From this viewpoint, data cannot easily included into each service and the dependencies between services and data needed to be defined.

So we defined the program group as a group of all the dependent programs related to each entry point. The entry point is the interaction point between the system and user, such as an API, a front-end screen, and a batch starting point. We used the program name or classes name as the name for entry points and grouped other programs by following the function or method calls, class definition or inheritance. When we reached the already called programs, we stopped following because the called programs were already included in the program group. On the next step, the data accessed from the program group are defined based on data accessed from the programs that are included in the program group. Figure 2 show the process of extracting the dependency information.

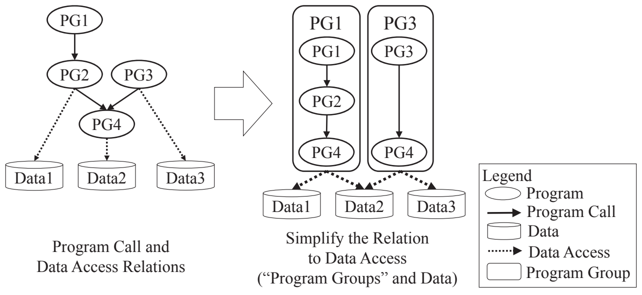
PG1
PG2
PG3
PG4
Data1
Data2
Data3
Program Call and Data Access Relations
Simplify the Relation to Data Access ("Program Groups" and Data)
Legend
Program
Program Call
Data
Data Access
Program Group

Fig. 2. Focusing on the dependencies of data access

As described in Figure 2, when a program which is called from many entry points, it will be included into each program group and define the data access from each program group (e.g. "PG4" in Figure 2). In this step, we focused on creating the dependency with data access by including all the program calls inside the program groups. Through our industrial experience and observations, we found that the structures of enterprise systems tended to be aligned to the frameworks such as the three-tier architecture. In such architecture, entry points are defined as a program and other programs are called to execute the implemented logic and access necessary data.

C. How to extract the dependencies from the source code

We use static analysis as analyzing the source code without execution to collect the dependencies which are program calls and data access. It was less expensive for us than collecting the dynamic behavior by executing the industrial system because some system needed to be modified to capture the behavior. Our method does not limit to static analysis as we use the information of program calls or data access which can be collected by dynamic analysis.

In programs implemented in a procedure language such as COBOL, the structure of function calls is quite simple as the main program is defined and it calls the sub programs. In case of object-oriented program, such as Java, element that is the main program may not be obvious and should be specified by other information such as user documents and framework structure. We conducted two case studies which are Java and COBOL. The elements and dependencies extracted are summarized in Table I. Java has both method calls and other relationships such as inheritances. We also captured the inheritance of class as relationship from the child class to the parent class as when the child class is called, the parent class can be related. In the case studies, we describe how we extracted the programs, data, and the relationships in detail on a practical example. When there are other elements and relationships which should be captured such as stored data accessed in other means such as network storage or other systems which needs to be considered, the relationship as data access from the entry point to the network storage can be added.

TABLE I. EXTRACTION OF ELEMENTS AND DEPENDENCIES

Elements	How we extracted from Java web application	How we extracted from COBOL application
Entry points	Classes with "@Control" annotations	Main program
Data	Class with "@Entity" annotations Description written in "@Table" annotation	DB tables, Files
Program Calls	Method calls, Class definition, Class inheritance (child to parent)	Module calls by CALL statement
Write Access to Data	Method which has particular phrase("set" or "add") in its name called in the class which is defined as data (e.g. setName, addElement)	Table or File creation, updates and deletion
Read Access to Data	Method called and class instantiated in the class which is defined as data excluding the case of "write Access" (e.g. getName)	Table selection or File reference

D. Clustering using the relationship between program groups and data

We now have program groups which have no program calls among them as all called programs are included in each group. The relationships between program groups are defined with data access dependencies. We also hypothesize that data accessed by few programs groups can be grouped together with the program groups. It is recommended in the loosely coupling architecture that local data should be encapsulated with the related process. We were inspired by this idea and we decided to allocate the data to the closely related program. We focus on the data which constitute tables or files stored in the system. We analyzed the relationship between program groups and data, and classified them as "read access" (such as referring the data) and "write access" (such as creating, updating or deleting the data) [10]. We employed the software clustering to program groups and data with the dependencies of data access to create the candidates. Our method uses the SArF [9] clustering technique, which applies graph clustering to the static program dependencies and the dependencies between a program and data [10]. However, the proposed method shown later doesn't necessarily need SArF itself. A clustering algorithm that can gather the strongly related programs and data of software might be applied instead of SArF clustering.

E. Clustering and visualization technique (SArF Map)

SArF is a clustering technique to decompose the system for comprehension [9] and visualize the whole structure using a city metaphor [11]. The procedure of the SArF Map technique comprises the following steps [11]: (1) Perform software clustering on the target software system using a SArF clustering algorithm, which gathers features in the resulting clusters. The output is a dendrogram. (2) Generate the abstract tree model of features from the dendrogram. (3) Lay out the buildings (classes) in the city block for each feature cluster). The layout algorithm reflects the layer of each class on its position. (4) Lay out features. Blocks of relevant features are closely placed as possible. (5) Overlay information used for the analysis on the blank map (e.g. assign package information as the color of each building).

The data, which are closely related to the program groups can be grouped together; however when many program groups share the same data, the relationship can be weakened. This concept matches the dedication score in SArF [9] as it calculates the relationship to be stronger when the modules access only one module but weaker when many modules access the module (e.g., logging functions are weakly related). In addition, we weighed write access stronger than read access since the idea that program groups that write to the business data have a strong relationship with business data. Thus, the dedication score for our method is derived from the (1) of [9] as follows:

$$D(A, B) = 1 / (n \ d\_in (B)) \qquad (1)$$

where A is the program group, B is the data accessed, d_in(B) is the number of program groups accessing data B, and n is the constant number to weigh data access. SArF uses the input of dependencies information between program modules, but our method uses the information between program groups and data. We adopted the approach in the SArF as creating the pseudo-member for "read" and "write" for each table to calculate the dedication score, which lowers the weights of trivial data accesses such as "read" accesses to master data (Section III.A of [10]). In addition, we weighed the write access to be double of the read access, so that in (1), the value of n are 1 when data is written, and 2 when data is read, respectively.

F. *Splitting off the common data*

Our previous work [10] shows that software clustering with data accesses can be used to reveal the characteristics of the database tables and this technique is now applied to identify omnipresent modules in the context of this paper, which is referred here to as common data. However the threshold for the common data is not defined and it is still not clear whether there is a data access left between the clustering results. To make the data access clear, we focus on the write operation between clusters and master data access. We classify the following two types of data as common data.

(1) Data which are written from the program group outside of the boundaries.

(2) Data which are read only, i.e., with no write operation from any program group.

We check the data access information from all the program groups whether data are written from the outside of the boundaries (i.e., program groups in other clusters) or not written from any program group within the user defined programs. When the data are classified as common data, we exclude them from the clustering results, and break up the program groups or data which have no relationships into smaller groups. An example of extracted candidates of microservices is shown in Figure 3. wherein data2 and data4 are classified as common data.

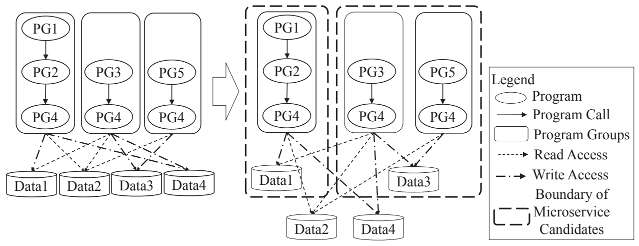

Fig. 3 Extracting the candidates of microservices

G. *Visualizing the clustering results*

We aim to serve our results as one of the draft proposals. Our results need to be reviewed by a software developer to finally decide how and which part of the system should be defined as microservices. To support the review by the developers, we visualize the extracted results using SArF Map [11] . The visualization is based on the SArF algorithm, and it places areas with a strong mutual relationship close together. We describe the candidates of microservices as the area surrounded by "red lines" on the map (Figure 4). Figure 4 and Table I show the output of our method which visualizes the extracted candidates of microservices.

TABLE II. ENTITY AND CONCEPT MAPPED IN THE VISUALIZATION

Elements	Metaphor in visualization
Program groups	Building (in color except black)
Data (tables /files)	Building colored in black
Extracted candidates of microservices	Block surrounded with red line (boundary)
Extracted candidates of microservices which is completely independent	Block surrounded with blue line (boundary)
Relevance between extracted candidates	Distance between blocks
Write data access relationship	Link representation as lines

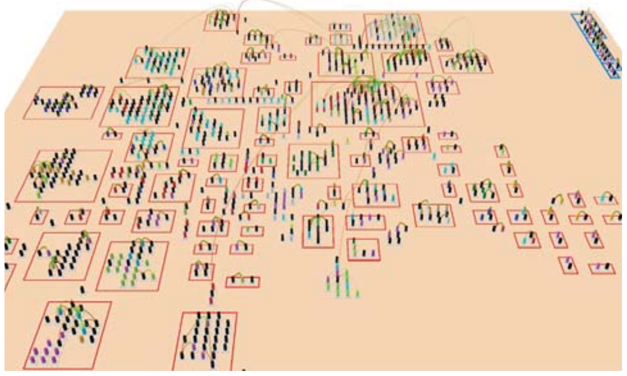

Fig. 4 Visualization of extracted candidates of microservices as red and blue blocks

H. Use cases of our output

In our visualization, developers can select the extracted candidates to see which program groups and data are grouped together in the block surrounded by red lines. They can also decide which part to analyze in detail, i.e., add information of dynamic analysis to decompose from the existing monolithic application. The relationship left between the candidates needs further analysis and our analysis can narrow down the relationships of especially writing data between candidates to common data. Data inside the red line will be written only from the program groups inside the block.

I. About the common programs

We deployed common programs into each program group, although when the extracted candidates have program groups with same common program, the common program can be merged as in Figure 5.

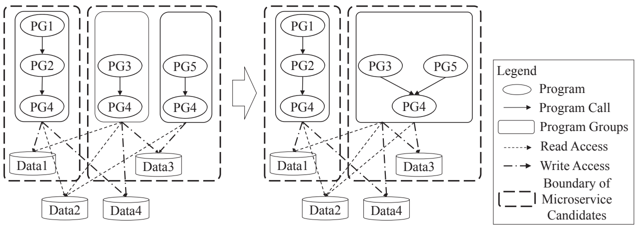

Fig. 5 Common programs in the candidates

It has been reported that in industrial service computing, services that are typically involved are dedicated to relatively narrow and concise tasks [12]. We hypothesize that most programs are dedicated to tasks and common programs are the small parts of the system which will be duplicated into several program groups when we apply our method to industrial systems. When the programs are finally included into two or more program groups, methods to manage the program code should be investigated. The code can include logics which are not executed when the input from the caller is limited and the code can be separated to each candidates.

IV. The Focus of the Case Studies

To evaluate the effectiveness of our method, we set up the following questions for the case studies. The questions are based on the viewpoint of microservices [5], [6], as each service should include core function and should be loosely coupled.

FS1: Are the extracted candidates of microservices valid? That is, do they match to the design by developer and include the necessary function?

FS2: Are the extracted candidates loosely coupled? That is the number between candidates are fewer than other results?

We conducted case studies in two applications to address these focus points.

V. Case Study

A. Case study 1 (Spring Boot Pet Clinic)

First we applied our method to an open source application to demonstrate how our method works. We applied our method to "Spring Boot Pet Clinic" because this application has both the monolithic application[1] and the microservices version[2], which can enable us to validate our results. The specification is given in Table III.

TABLE III. Code specification of the case study

Type of information	Data
Application	Spring Boot Pet Clinic (Downloaded on Feb. 16, 2018)
Programming language	Java
Lines of code	630
Number of Java class files	25

1) Extracting the dependencies

First, we specified the entry points. This application was implemented on spring boot, which is a framework for web application for Java. The entry points should be the point which will be the front-end or API to be defined. In this case, we selected the class which has "@Control" annotation because these classes contain user control which would be close to front-ends.

In this application, four entry points were defined (Table IV). After we defined the entry points, we defined the data using the class annotations. Java persistence API, which is a Java application programming interface specification that describes the management of relational data in applications in Java, was used with annotations. In this case we extracted the class with "@Table" as data and defined the name of data as the value of name attribute. For example the "Pet" class was defined with "@Entity" and "@Table" (Figure 6). The annotation "@Table" has "pets" in its name attribute of name so we defined this class as data.

```
@Entity
@Table(name = "pets")
public class Pet extends NamedEntity {
…
 public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public Date getBirthDate() {
        return this.birthDate;
    }
…
}
```

Fig. 6 Source code example of data object

[1] https://github.com/spring-projects/spring-petclinic
Downloaded at 17:06 on February 16, 2018 (Last Commit is on Jan 20, 2018).

[2] https://github.com/spring-petclinic/spring-petclinic-microservices
Downloaded at 17:06 on February 16, 2018 (Last Commit is on Jan 20, 2018).

After we defined the data, we created the relationship between the entry points and the data. We started from the entry point and followed the method calls until it reaches the data. In this case, we analyzed the read/write access by using the name of the method. When the name has a particular phrase, in this case as "set" or "add", it is usually setting or adding data, which can be captured as a "write" access. The method can be named by the developer but there are some naming conventions defined for data objects such as JavaBeans [13], and we rely on the method name of the method for classifying data access. For instance as in Figure 7, we started from the entry point "PetController" and followed the method that is calling. In this case, in the method "initCreationForm", the class definitions of "Pet" and "addPet" in class "Owner" is called.

```
@Controller
@RequestMapping("/owners/{ownerId}")
class PetController {
…
    @GetMapping("/pets/new")
    public String initCreationForm(Owner owner, ModelMap model) {
        Pet pet = new Pet();
        owner.addPet(pet);
        model.put("pet", pet);
        return VIEWS_PETS_CREATE_OR_UPDATE_FORM;
    }
```

```
Caller:
org.springframework.samples.petclinic.owner.PetController
#initCreationForm
Callee (Methods called):
org.springframework.samples.petclinic.owner.Pet#Pet
org.springframework.samples.petclinic.owner.Owner#addPet
org.springframework.ui.ModelMap#put
```

Fig. 7 Source code and the relationship extracted

The Pet and the Owner are both the data as they have "@Table" annotations. The class definition is defined as read access and the "addPet" is defined as write access because of the method name. The name without defined phrase is classified as read access in this case study. After this process, we obtained the relationship of entry point, which is the same as program groups, and data as in Figure 8.

```
org.springframework.samples.petclinic.owner.PetController,pets#read
org.springframework.samples.petclinic.owner.PetController,owners#write
```

Fig. 8 Example of Extracted Relationship of Program Groups and Data

TABLE IV. ELEMENTS EXTRACTED IN THIS CASE STUDY

Elements	Extracted Value
Entry points (extracted the classes with "@Control" annotations)	`org.springframework.samples.petclinic.owner.VisitController` `org.springframework.samples.petclinic.owner.PetController` `org.springframework.samples.petclinic.owner.OwnerController` `org.springframework.samples.petclinic.vet.VetController`
Data ("@Table" annotation)	`visits, pets, owners, vets, vet_specialities`

We created the program group by following the method call including class definition. In the case of class inheritance, we extracted the relationship as a child class to a parent class to capture the data access from the parent class. The extracted elements in this case study are given in Table IV.

2) *Clustering and visualization*

We assigned the relationship of the program groups and data, described in Figure 8 as the input and applied software clustering using SArF and also checked whether there are common data or not. In this case, the common data are not extracted. We finally obtained the results described in Figure 9. The results show us that the application can be decomposed into three candidates of microservices, which is "VisitController", "PetController/OwnerController", and "VetController". The "VetController" is completely independent as it has no relationship between other classes (blue block). Data inside the red block are not written (updated) by the process outside the block. For visit, pet, and owner, there are relationships between them however "Pets" and "Owner" are extracted together and the "visit" as another candidate.

TABLE V. ENTITY AND CONCEPT MAPPED IN THE VISUALIZATION

Elements	Metaphor in visualization
Entry points (program groups)	Building (in color except black)
Data (tables /files)	Building colored in black
Extracted candidates of microservices	Block surrounded with red line (boundary)
Extracted candidates of microservices which is completely independent	Block surrounded with blue line (boundary)
Relevance between extracted candidates	Distance between blocks
Data access relationship	Link representation as lines

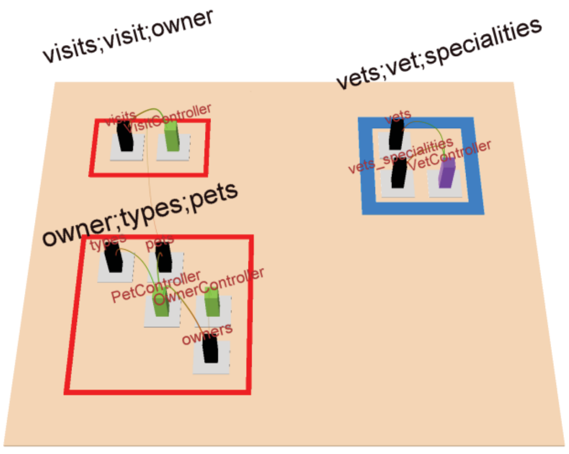

Fig. 9 Visualization of extracted candidates of microservices as red and blue blocks (our results)

3) Evaluation by Comparing with the "petclinic microservice"

As previously mentioned, the "Spring Boot Pet Clinic" application has both the monolithic application and the microservice version. To evaluate the results, we focused on whether which functions are grouped. The microservice version has the **Customers, Vets and Visits Service**s defined. The Customers Service contains both the owner and the pet information, the function to manage "Pet" and "Owner" as "PetResource" and "OwnerResource" are grouped together. The list of the programs in each service are in Table VI. In our results, "PetController" and "OwnerController" are grouped together and "Visits" and "Vets" are separated. From this result, our result as the candidates of microservice and matches the function included in the microservice version, which developer thinks that the function to be combined (FS1 is supported).

There are also some difference between our results and the microservice version. Our results include some files such as "BaseEntity.java" or "Pet.java" which are common programs. These programs needed further analysis whether it is needed to be assigned to two candidates (1 and 2 in Table VI) for example whether the logic can be separated.

TABLE VI. EXTRACTED CANDIDATES AND DESIGNED MICROSERVICES

	Source code in extracted candidates of microservices	Source code defined in the "Spring Boot microservice"

1	**`OwnerController.java`**
	`Person.java`
	`Owner.java`
	`OwnerRepository.java`
	`PetController.java`
	`Pet.java`
	`PetType.java`
	`BaseEntity.java`
	`NamedEntity.java`
		`OwnerResource.java`
		`Owner.java`
		`OwnerRepository.java`
		`PetResource.java`
		`Pet.java`
		`PetRepository.java`
		`PetType.java`

2	**`VisitController.java`**
	`VisitRepository.java`
	`PetRepository.java`
	`Pet.java`
	`BaseEntity.java`
		`VisitResource.java`
		`Visit.java`
		`VisitRepository.java`

3	**`VetController.java`**
	`Vet.java`
	`VetRepositry.java`
	`Vets.java`
	`(vet_specialities)`
		`VetResource.java`
		`Speciality.java`
		`Vet.java`
		`VetRepository.java`

TABLE VII. MUTUAL DEPENDENCIES BETWEEN CANDIDATES

	Extraction method	Number of candidates	Read access	Write access	Sum of access
1	Grouping by name (design)	4	2	1	3
2	Our method	3	1	0	1
	Percentage of 2/1	75%	50%	0%	33%

To see how the extracted candidates are loosely coupled, we compared with other types of decompositions. One of the decompositions is to use "nouns or resources". When the entry points and the data have same name or name related, it can be merged, e.g., "VisitController" and "visit" data are related and can be grouped. The naming convention are used to manage the files. In this case, we had four candidates, with each "controller" for each service. In our case, the dependencies between candidates are data accesses; therefore considering the number of dependencies between candidates, we compared our method with others. The number between candidates are in Table VII. and as seen in table, the number between the candidates are fewer than by grouping by name. (FS2 is supported).

4) Comparison with SArF with data access

We tried previous software clustering tools. SArF Map [9], [11] visualizes software architecture from feature and layer viewpoints. It has been extended to visualize the data access [10]. To capture the "feature components", the buildings in this map are class/source files and data (Figure 10). All "Controllers" are extracted in the separate component as blocks. Other methods are needed to consider which blocks can be combined and our method can be applied for this purpose. A brief list of the source codes in the extracted blocks is given in Table IX.

TABLE VIII. ENTITY AND CONCEPT IN SARF WITH DATA ACCESS

Elements	Metaphor in visualization
Class/ source file/ data(as table)	Building
Package	Color of the building
Feature (cluster)	City block
Layer	North-south way
Relevance between features	Street and distance
Relationship between classes	Link representation as lines
Classes with no relation	Block surrounded with blue line

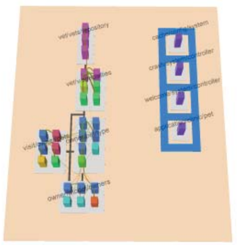

Fig. 10 Visualization by SArF [9], [11] with Data Access [10]

TABLE IX. LIST OF SOURCE CODES BY SARF WITH DATA ACCESS

	Partial list of source codes for each extracted block
1	**`OwnerController.java`**`, BaseEntity.java, etc.`
2	**`PetController.java`**`, PetType.java, PetRepository.java, etc.`
3	**`VisitController.java`**`, Pet.java, etc.`
4	`Vet.java, (vet_specialities), etc.`
5	**`VetController.java`**`, VetRepository.java, etc.`

B. Case study 2 (Industrial Application)

We applied our method to an industrial application from the purchasing operations department in Fujitsu and the specifications of the application are given in Table X. The system was migrated about 12 years ago into the current system, and the initial design was much older. In this case study, we interviewed three leading developers who knew about the business and the specification of the application.

TABLE X. CODE SPECIFICATION OF THE CASE STUDY

Type of information	Data
Application	Industrial application for purchase operation
Programming language	COBOL
Lines of code (KLOC)	2,031 K
Number of COBOL files	2,269

1) Extracting the dependencies

For analysis, the entry points are not clearly defined; therefore we selected as the entry points as the programs that are not called from other programs within the user-defined COBOL program. These programs should be the main programs which are called from the outside of the system (such as frameworks and batch programs). For data access, we weighed the write access to be double of the read access, so that in equation (1) the values of n are 1 when data are written, and 2 when data is read respectively. We classified data access information as read and write operations which includes creating, updating, and deleting operation from the embedded SQL statement. The number of entry points and data are given in Table XI.

TABLE XI. NUMBER OF ELEMENTS EXTRACTED IN THIS CASE STUDY

Elements	Number
Number of entry points (extracted “main programs”)	533
Data (DB tables)	673

2) Clustering, splitting off the common data, and visualization

The visualization of case study 2 result is given in Figure 4, which is in the previous section. The number of elements extracted in this case study are given in Table XII.

TABLE XII. THE NUMBER AND SIZE OF EXTRACTED CANDIDATES

Descriptions	Data value
Number of extracted candidates of microservices	199
Size (lines of code) of extracted candidates (Average)	16,879
Size (lines of code) of extracted candidates (Median)	5,630

The system was decomposed into 199 candidates with an average size (lines of code) of 17 K steps (line of code) and at a median of 5 K steps. Furthermore, 533 program groups were extracted as the minimum unit, which contained programs in about 6.68 feature components of SArF in average. SArF extracts “feature components” as the unit of implementation and our results extract several feature components combined.

3) Evaluation by review from developers

In this case, we could not obtain the “microservice version” of this application. To evaluate our results, we asked developers to verify whether our extracted candidates match the function in business. The candidates were selected because there were many program updates caused by specification changes and these may be reasonably partitioned from the system. From 199, we selected 5 which were related to the function of “managing payment” to be observed in greater detail by the developers. The developers remarked that four of the candidates were reasonable as one small service, such as programs using data generated by other grouped programs. For example, the user’s input was checked by the first program and the following program used the checked results of the first program and generated a form for purchases and these programs and data are grouped together. However, one of the candidates appeared to be mixed. Program groups that update nearly all files that appear in the candidates were not executed together with the other programs groups in the candidates. They were utility programs for initialization or maintenance which update almost all the files. These programs may need to be excluded as they do not run in the business hours. Further information is needed in the case of candidates; however the targets of detailed analysis are narrowed by our analysis as the reviewed candidates were quite close to the business process of the grouped programs. (FS1 is supported)

To see how the extracted candidates are loosely coupled, we compared with other types of decompositions. In this case, the programs and data (database tables) were initially designed using a naming convention which was designed to classify the business units with their names. For example, the programs which are included in “manage payment” are named using “AA” as “AA000001.cob” for the file name. We grouped the program files and data having names that start with “AA” as the group to be compared. The results are given in Table XIII.

TABLE XIII. MUTUAL DEPENDENCIES BETWEEN CANDIDATES

	Extraction method	Number of candidates	Read access	Write access	Sum of access
1	Grouping by name (design)	71	3134	323	3457
2	Our method	199	2725	192	2917
	Percentage of 2/1	280%	86%	59%	84.4%

The result shows that the candidates obtained by our method reduced mutual dependencies to less than about 15% compared with the categories of naming convention ruled at the design phase (84.4% are left). We counted both mutual dependencies and common data access and the result are given in Table XIII. The write accesses between the candidates are fewer than that of the partition by using naming convention and they are only between common data, which is approximately 15.7% of the whole data. From this result, we can state that the extracted candidates are less coupled than those of the extraction using the naming convention (FS2 is supported).

4) Consideration on "common" programs and Data

We counted the programs allocated into two or more program groups as 295; these program can be referred as common programs. We hypothesize that common programs only comprise a small part and the number of programs is approximately 13% of the total number. Although further analysis is needed, we can currently state that the common programs constitute a small part of the total system. For the result of common data, we obtained 106 tables out of 673 (15.7%) as common data by our method while in the classification based on the system design and 163 tables are classified as common data. The developers confirmed that the results were reasonable because the difference between the two classification was due to the change of usage in maintaining the software for a long time as some common data by design were only accessed from one candidate, which should be classified as local data.

VI. THREATS TO VALIDITY

In the first case study, the application is an example to demonstrate the monolithic style and the microservice architecture style. In this case study, we compared the result by checking whether the key programs, such as "PetController" and "OwnerController" are grouped in the same candidates as "PetResource" and "OwnerResource" and are also in the same microservices. The comparison usually should be done quantitatively when evaluating clustering results [14], [15], [16]; however in our case, the programs in the extracted candidates and the microservice style application have some similar but different codes. Further investigation is needed to determine the applicability of our method to different applications.

The second case study was conducted on a practical industrial application although in this case, there was no "correct" microservice architecture style implementation for the target application. To evaluate the extracted candidates of microservices, three leading developers reviewed our results; however the number of reviewed candidates are limited. To mitigate this effect, the developers also viewed the whole structure with our visualization, and we discussed the whole structure with the developers.

VII. RELATED WORKS

In 1972, Parnas reflected on the criteria to be used in decomposing systems into modules [17]. Since then, functional decomposition has remained an important topic in software engineering. As software systems grew and became more complex, software engineers started to distribute modules and procedures over networks, e.g., as remote objects, components or web services. The common problem in these efforts is that it is a tedious manual effort to identify the unit of function, such as modules, procedures, objects, microservices, etc. it requires analysis of many dimensions of software architecture views and often heavily relies on the experience and expertise of the expert performing the extraction [6], [7]. Various approaches have been proposed to support developers to decompose a software system [2], [18], such as software architecture reconstruction [2], which is a reverse engineering approach that aims at reconstructing viable architectural views of software application. The researchers identified the major challenges of software architecture reconstruction as abstracting, identifying and presenting higher-level views from lower-level and often heterogeneous information [2].

Considering abstracting, identifying and presenting higher-level views software architecture from lower-level information, software clustering is one of the techniques to capture the high-level abstraction views of the system [2], [3], [9] . Mancordis et al. proposed "Bunch" for software structure recovery based on the dependencies between modules [19]. In this work, the problem with omnipresent modules is mentioned, which are the modules with many relations and which cause clustering results to be complicated. Kobayashi et al. proposed SArF [9], [11] to automatically weaken this effect by defining a dedication score to each module to overcome the complicated clustering results. SArF has also been extended to visualize data accesses [10]. Another approach is ACDC; an algorithm for comprehension-driven clustering [20] which is a pattern-based approach that utilizes several rules to cluster modules, e.g., clustering a dominator and its dominated modules. There are also works on clustering using a topic model [21]. These techniques are useful to find the core functions in programs in an implementation but when targeting re-modularization to loosely coupling software architecture, another view such as components-and-connector view is needed to partition into unit of executions [22], which is the necessary for defining services. Our method supports extraction of the candidates of microservices by analyzing the source code which is applicable to large legacy codes.

Considering the creation of microservices, Service Cutter [8] has been proposed to decompose a monolithic application based on 16 criteria presented by the authors. This process requires a manual extraction of building block of microservices as "nanoentities" and its relationships. Our method can support the extraction since as our results can be the candidates of microservices, which can serve as the building block. Levcovitz et al. proposed a decomposing monolithic system by analyzing the relationship between the entry point of the system, business functions, and data [23]. Each service can be defined by connecting the entry point, business logic and data. Their approach is quite similar to ours, but the methods of treating data used by many services are not defined. Our method defines the common data and also gathers services which use the same data to support corresponding decomposition. There are other works that also use data as the fundamental of decomposing the system into services. Chen et al. proposed using the Data Flow Diagram (DFD) to define the microservices [24]. This work defined the pattern of the DFD whether it is decomposable. The work can be useful when the DFD is defined. In our work the structure is first decomposed using program code information, which can be applied to generate information and also to visualize the results to determine the correctness of the dependencies extracted from the code.

Considering other forms of decompositions, there are works which have been proposed to extract the reusable software components from object-oriented APIs [25]; this work focuses on whether the components are called together by the API. Our works focus on how to define the microservices, and emphasize on how the data are accessed by the programs.

Considering comprehension and discussion of the migration and modernization, Khadka et al. pointed out that in the migration process involves identifying candidate services, service extraction and visualizing the extracted results for discussion such as matching with the knowledge of business with stakeholders [26]. Our works also visualize the whole structure to obtain the checkable output, similar to SArF [9], [11] which visualizes the clustering results as a city metaphor with vocabularies of the source code labeled on the map [27]; this is useful to stakeholders in comprehending the whole structure.

VIII. Summary

We developed a method that identifies the candidates of microservices from the source code in decomposing the existing system. We presented a case study on a small application and another case study on a practical industrial application with preliminary evaluation by developers. Our method can be used for initial analysis when selecting the part to be partitioned; however some information such as execution timing and revisions, need to be added to improve the candidates of microservices identification. We are currently applying our methods to many systems or languages to validate whether our method can be useful in extracting candidates of microservices. Furthermore, we currently seek for further steps so that in addition to grouping, we can also recommend how to modify the programs for microservices.

Acknowledgement

We would like to thank the members of SI Core Practice Division of Fujitsu for providing their support. We also appreciate the help of the developers for evaluating the results.

References

[1] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen and J. Hage: "How do professionals perceive legacy systems and software modernization?," *in Proc. of 36th Int. Conf. Softw. Eng. (ICSE) 2014*, pp.36-47, 2014.

[2] C. Birchall: "ReEngineering Legacy Software", Manning Publications Co. 2016.

[3] S. Ducasse and D. Pollet, "Software architecture reconstruction: a process-oriented taxonomy," *IEEE Trans. on Softw. Eng.*, Vol. 35, No. 4, pp. 573-591, 2009

[4] T. Erl, "Service-oriented architecture: concepts, technology, and design", Prentice Hall PTR, 2005.

[5] J. Lewis, and M. Fowler, "Microservices", [Online] https://martinfowler.com/articles/microservices.html [Jul. 6, 2018]

[6] S. Newman, "Building Microservices", O'Reilly media, 2015

[7] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," *in Proc. of 6th Int. Conf. Cloud Comput. Serv. Sci.*, no. January, pp. 137–146, 2016.

[8] M. Gysel, L. K¨olbener, W. Giersche, and O. Zimmermann, "Service Cutter: A systematic approach to service decomposition," *in Proc. of ESOCC 2016*. Springer, September 2016, pp. 185–200.

[9] K.Kobayashi, M.Kamimura, K.Kato, K.Yano, and A. Matsuo, "Feature-Gathering Dependency-Based Software Clustering Using Dedication and Modularity", *in Proc. of 28th IEEE Int. Conf. on Software Maintenance (ICSM)* 2012, pp.462-471, 2012.

[10] K. Yano and A. Matsuo, "Data Access Visualization for Legacy Application Maintenance," *in Proc. of Int'l Conf. on Software Analysis and Evolution and Reengineering* (*Saner) 2017*, pp. 546–550, 2017.

[11] K.Kobayashi, M.Kamimura, K.Yano, K.Kato, and A.Matsuo, "SArF map: Visualizing Software Archtecture from Feature and Layer Viewpoints". *in Proc. of 21st IEEE Int. Conf. on Program Comprehension (ICPC) 2013*, pp.43-52, 2013.

[12] G. Schermann, J.Cito, P. Leitner, "All the Services Large and Micro: Revisiting Industrial Practice in Services Computing" *In: A. Norta, W. Gaaloul, G. Gangadharan, H. K. Dam (eds) Service-Oriented Computing – ICSOC 2015 Workshops*., pp. 36-47, 2015. Lecture Notes in Computer Science, vol 9586. Springer, Berlin, Heidelberg, 2016.

[13] Oracle Coopoerations: JavaBeans, [Online] https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html [Jun. 1, 2018]

[14] V. Tzerpos and R. C. Holt "MoJo: A distance metric for software clusterings", *in Proc. of Working Conf. on Reverse Eng., (WCRE) 1999*, pp. 187-193, 1999.

[15] Z. Wen, and V. Tzerpos, "An effectiveness measure for software clustering algorithms," *in Proc. of Int'l Workshop on Prog. Compre., (IWPC) 2004*, pp. 194-203, 2004.

[16] M. Shtern and V. Tzerpos, "Lossless comparison of nested software decompositions," *in Proc. of Working Conf. on Rev. Eng., (WCRE) 2007*, pp. 249-258, 2007

[17] D. L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM Volume 15 Issue 12*, pp. 1053-1058, Dec. 1972.

[18] I. Candela, G. Bavota, B.Russo, R. Oliveto: "Using Cohesion and Coupling for Software Remodularization: Is It Enough?", *ACM Transaction on Software Engineering and Methodology*, Vol. 25, No. 3, Article 24, May 2016.

[19] 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 Proc.of IEEE Int. Conf. on Software Maintenance. - 1999 (ICSM'99)*., pp. 50–59, 1999.

[20] V. Tzerpos and R. C. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," *in Proc. of Working Conf. on Reverse Eng., (WCRE) 2000*, pp. 258-267, 2000.

[21] S. Ducasse, T. Girba, and A. Kuhn, "Distribution map" *in Proc. of IEEE Int. Conf. on Software Maintenance (ICSM) 2006*, pp. 203-212, 2006.

[22] Paul Bachmann, Felix Bass, Len Garlan, David Ivers, James Little, Reed Merson, Paulo Nord, Robert Stafford, Judith Clements, "Documenting Software Architectures: Views and Beyond", Addison-Wesley Professional, 2nd Edition, 2010.

[23] A. Levcovitz, R. Terra, and M. Tulio Valente, "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems," *3rd Brazilian Work. Softw. Vis. Evol. Maint*., October, pp. 97–104, 2015.

[24] R. Chen, S.Li, Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach, *in Proc. of Asia Pacific Software Engineering Conf. (APSEC) 2017*, pp466-475, 2017.

[25] A. Shatnawi, A. D. Seriai, H. Sahraoui, and Z. Alshara, "Reverse engineering reusable software components from object-oriented APIs," *J. Syst. Softw.*, vol. 131, pp. 442–460, 2017.

[26] R. Khadka, A. Saeidi, S. Jansen, J. Hage, and G. P. Haas, "Migrating a large scale legacy application to SOA: Challenges and lessons learned," *in Proc. of Work. Conf. on Reverse Eng. (WCRE) 2013*, pp. 425–432, 2013.

[27] K. Yano and A. Matsuo, "Labeling Feature-Oriented Software Clusters for Software Visualization Application," *in Proc. of Asia-Pacific Software Engineering Conference (APSEC), 2015*, pp. 354–361.