Re-architecting OO Software into Microservices
A Quality-Centred Approach

Anfel Selmadji(✉), Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane

LIRMM, CNRS and University of Montpellier, Montpellier, France
{selmadji,seriai,bouziane,dony}@lirmm.fr,
rahina.oumarou-mahamane@etu.umontpellier.fr

Abstract. Due to its tremendous advantages, microservice architectural style has become an essential element for the development of applications deployed on the cloud and for those adopting the DevOps practices. Migrating existing applications to microservices allow them to benefit from these advantages. Thus, in this paper, we propose an approach to automatically identify microservices from OO source code. The approach is based on a quality function that measures both the structural and behavioral validity of microservices and their data autonomy. Unlike existing works, ours is based on a well-defined function measuring the quality of microservices and use the source code as the main source of information.

Keywords: Object-Oriented · Microservices
Migration · Identification

1 Introduction

Recently, microservice architectural style has become an essential element for the development of applications deployed on the cloud or for those adopting the DevOps practices [5,10]. In this style, an application consists of a set of small services which are independently deployable. Usually, each microservice can only manage its own data [10,12]. These services communicate through lightweight mechanisms and they are deployed using containers such as Docker [12–14].

For the cloud, microservices facilitate the reconfiguration of an application according to the changes that may occur at runtime [3]. These changes can be related to cloud resources (e.g. resource allocation, etc.), quality of service (e.g. scalability guarantees, etc.) or any other event (e.g. failure, etc.). For DevOps, microservices facilitate a continuous integration, delivery and deployment tasks [5].

Besides the adoption of microservice architectural style for the development of new applications, the migration of existing monolithic ones to this style allows

K. Kritikos et al. (Eds.): ESOCC 2018, LNCS 11116, pp. 65–73, 2018.
https://doi.org/10.1007/978-3-319-99819-0_5

them to benefit from all the above-mentioned advantages. The migration process consists of three steps: (1) comprehension of the existing system (i.e. reverse engineering), (2) identification of microservices and (3) packaging them.

Different works have proposed strategies to achieve this migration [4,8,9,11]. Some of these approaches propose limited and ad-hoc heuristics for identifying microservices. Indeed, they do not consider the data autonomy of microservices [4,8] or they focus on measuring internal coupling and cohesion of microservices and not their external coupling [4,11]. Finally, some others require, in addition to the source code, the use of other artifacts [8,11] that can be unavailable.

In this paper, we tackle these limitations by proposing an automatic approach to identify microservices from OO source code. Unlike existing approaches, ours considers the specificity of microservices. It is based on a quality function that measures the functional validity of a microservice and its data autonomy.

The remainder of this paper is organized as follows. Section 2 outlines related works. Section 3 presents the proposed approach for identifying microservices. Section 4 evaluates our proposal. Finally, Sect. 5 concludes the paper.

2 Related Works

An attempt at providing a structured approach to identify microservices from monolith is Service Cutter [8]. Service Cutter uses artifacts and documents related to the software engineering process to build a graph representation which is decomposed through graph cutting. The limitation of Service Cutter is that the used artifacts and documents can be unavailable or not up to date.

Levcovitz et al. [9] proposed an approach to identify microservices from monolithic enterprise applications. The approach consists of grouping each user interface with the business functionality it calls and the database tables used by this functionality in a microservice. Therefore, the main limitation of this approach is that it is based on a restrictive hypothesis about the architecture of the monolith to be decomposed (i.e. MVC architecture).

Mazlami et al. [11] proposed a formal model to extract microservices from monoliths. More precisely, the authors used three formal coupling strategies and embed those in a graph-based clustering algorithm. In this approach, some coupling strategies depend on the change history of the code. Thus, if it is unavailable or consists of a limited number of commits, the approach is unusable.

Baresi et al. [4] proposed an approach to identify microservices from an OpenAPI specification. The identification process consists of matching the terms in the specifications against a reference vocabulary to suggest possible decompositions. The limitation of this approach is relying on well-defined interfaces that provide meaningful names. Moreover, database partitioning was not handled.

In conclusion, the existing works suffer from considerable limitations in terms of the restrictions associated with the used artifacts, the exploited information, and the partitioning measures on which they are based.

3 Microservices Identification from OO Source Code

Our microservices identification approach is based on four principles: (1) It considers OO software, (2) It exploits, mainly, the source code to identify microservices, (3) It defines a function that measures the quality of a microservice. (4) It exploits the information concerning the relations between the entities of the code and the information related to the persistent data manipulated in this code.

3.1 From Microservices Characteristics Description to Characteristics Evaluation

Characteristics of Microservices: Lewis and Fowler [10] define microservices as *small* services, communicating with lightweight mechanisms. These services are *independently deployable* by *fully automated deployment* machinery. They may be written in *different programming languages* and use *different data storage technologies*. Newman [12] considers microservices as *small*, *autonomous* services that work together. Pujals [13] defines microservices as autonomous lightweight processes, created and deployed with relatively small effort and ceremony.

Based on these definitions and others [14], we identified microservice's main characteristics: *(1) Small and focused on one function:* a microservice is typically responsible for a simple business functionality. *(2) Autonomous:* microservices are separate entities. They communicate via network calls and each one manages its own database. *(3) Technology neutral:* with a system composed of a set of microservices, each one can use different technologies. *(4) Automatically deployed:* if the number of microservices increases, automatic deployment is required.

The above characteristics can be classified into two categories: (1) those related to the structure and behavior of microservices and (2) others related to the microservice development platform. Therefore, to measure the quality of candidate microservices, only the characteristics that define microservice structure and behavior are selected: *small and focused on one function* and *autonomous*.

Evaluation of the "Focused on One Function" Characteristic: In our approach, a microservice M is viewed as a set of classes collaborating to provide one function. This collaboration can be determined from source code through the internal coupling, that represents the degree of direct and indirect dependencies between classes. Moreover, it can be determined by the number of volatile data[1] whose use is shared by these classes. It reflects the internal cohesion. Thus, *FOne* (*Eq.* 1) evaluates the characteristic *Focused on One Function*.

$$FOne(M) = \frac{1}{2} \quad (InternalCoupling(M) + InternalCohesion(M)) \qquad (1)$$

[1] Attributes are an example of volatile data.

Evaluation of the Structural and Behavioral Autonomy of a Microservice: Microservices are separate entities. Thus, in order that a set of classes represents a microservices their dependencies on external classes should be minimal. This can be measured using external coupling (see *Eq.* 2).

$$FAutonomy(M) = ExternalCoupling(M) \tag{2}$$

Evaluation of the Data Autonomy of a Microservice: A microservice can be completely data autonomous if it does not require any data from other microservices. In order that a microservice require less external data, its classes need to manipulate the same data. Thus, FData (*Eq.* 3) is based on measuring data dependencies between the classes of the microservice ($FIntra$), and their data dependencies with classes not belonging to the microservice ($FInter$).

$$FData(M) = \frac{1}{2} \quad (FIntra(M) - FInter(M)) \tag{3}$$

The $FIntra$ (resp. $FInter$) function applied on a microservice M represents the ratio between the number of data shared between its classes (resp. with other classes) and the total number of data manipulated in the microservice.

Global Evaluation of a Microservice: The global evaluation (*Eq.* 4) of a microservices depends on the evaluation of its characteristics.

$$FMicro(M) = \frac{1}{n} \quad (\alpha FOne(M) - \beta FAutonomy(M) + \gamma FData(M)) \tag{4}$$

Where M is a microservice, α, β and γ are coefficient weights determined by software architect and $n = \alpha + \beta + \gamma$. By default, the value of each term is 1.

3.2 Evaluation of Microservice Characteristics Based on Metrics

Internal Coupling: Internal coupling evaluates the degree of direct and indirect dependencies between classes. The more two classes use each other's methods the more they are coupled. Hence, the internal coupling is measured as follows:

$$InternalCoupling(M) = \frac{\sum CouplingPair(P)}{NbPossiblePairs} \tag{5}$$

Where $P = (Cl1, Cl2)$ is a pair of classes of the microservice M, *NbPossiblePairs* is the number of possible pairs of classes in M, whereas *CouplingPair* is:

$$CouplingPair(Cl1, Cl2) = \frac{NbCalls(Cl1, Cl2) + NbCalls(Cl2, Cl1)}{TotalNbCalls} \tag{6}$$

Where *NbCalls(Cl1,Cl2)* is the number of calls of the methods of *Cl1* by those of *Cl2* and *TotalNbCalls* is the number of method calls in the application.

Indeed, measuring internal coupling using *Eq.* 5 takes into account the frequency of calls. However, it does not promote clusters in which all the classes are coupled. For this reason, we introduced the sum of the standard deviations between the coupling values in the evaluation of the internal coupling (*Eq.* 7).

$$InternalCoupling(M) = \frac{\sum CouplingPair(P) - SumStandardDev}{NbPossiblePairs} \tag{7}$$

External Coupling: External coupling evaluates the degree of direct and indirect dependencies of the classes belonging to a candidate microservices on external classes. It is measured similarly to internal coupling, with only one difference which is the set of used pairs. To measure external coupling, each pair consists of two classes such that exactly one of them belong to the microservice.

Internal Cohesion: Internal cohesion evaluates the strength of interactions between classes. Usually, two classes are more interactive if their methods work on the same attributes. Thus, internal cohesion is measured as follows:

$$InternalCohesion(M) = \frac{NbDirectConnect}{NbPossibleConnect} \tag{8}$$

Where *NbPossibleConnect* is the possible number of connections between the methods of the classes belonging to the microservice *M*, whereas *NbDirectConnect* is the number of connections between these methods. Two methods *m1* and *m2* are directly connected if they both access the same attribute or the call trees starting at *m1* and *m2* access the same attributes. Because our aim is to measure the cohesion between the classes of the microservices, the connections between the methods of the same class are not considered. Note that, this internal cohesion measurement metric is a variation of the metric TCC (Tight Class Cohesion) proposed by Bieman and Kang [7].

3.3 Clustering Process

To identify microservices from OO code, classes are grouped based on their dependencies. Hence, a hierarchical agglomerative clustering algorithm [1] is used. We consider our function to measure the quality of a microservice as the similarity function used in the algorithm. Thus, the classes that maximize the value of the quality function are grouped together. More details can be found in [1].

4 Experimentation and Validation

4.1 Research Questions and Data Collection

To validate our approach we conducted an experiment to answer the following research questions: **RQ1:** does the proposed approach produce an adequate

decomposition of an OO application into microservices? **RQ2:** is the definition of the quality function, without considering data autonomy, adequate? **RQ3:** does the evaluation of data autonomy enhance the quality of microservices?

To answer these questions, we have experimented on three OO applications of different sizes: small (*FindSportMates*[2]), average (*SpringBlog*[3]) and relatively large (*InventoryManagementSystem*[4]). Table 1 provides some metrics on them.

Table 1. Applications metrics

Application	No of classes	No of classes representing database tables	Code size (LOC)
InventoryManagementSystem	104	19	13447
FindSportMates	17	5	785
SpringBlog	42	5	1615

4.2 Experimental Protocol

The answers to the research questions are based on a tool developed in Java. To answer ***RQ1***, we used our tool to identify microservices. Then, we compared them to those identified manually. The protocol for answering ***RQ2*** is similar to the one used for ***RQ1*** with one difference: we set our tool to identify microservices based on a function related to the characteristics *"focused on one function"* and *"structural and behavioral autonomy"*. To answer ***RQ3***, we compare the precision and recall values related to the answers of ***RQ1*** and ***RQ2***.

4.3 Direct Results

The source code of each of the previous applications was partitioned into a set of clusters. Table 2 shows the results obtained based on the entire quality function (*FMicro*) and on a quality function without the data autonomy part (*FSem*).

Table 2. Microservice extraction results

Application	No of microservices		Average no of classes per microservice	
	FMicro	*FSem*	*FMicro*	*FSem*
InventoryManagementSystem	10	9	8.5	9.44
FindSportMates	2	2	6	6
SpringBlog	4	4	9.25	9.25

[2] github.com/chihweil5/FindSportMates.
[3] github.com/Raysmond/SpringBlog.
[4] github.com/gtiwari333/java-inventory-management-system-swing-hibernate-nepal.

To evaluate the microservices obtained by our approach, we compare them with those identified manually. Thus, we classify the microservices obtained manually in three categories: (1) Those that exactly match the ones identified by our approach. The microservices identified by our approach and are classified in this category are considered excellent. (2) Those that can be obtained by a simple composition/decomposition of the microservices identified by our approach. The microservices identified by our approach of this category are considered good. (3) Those that are neither in the first nor in the second categories. The microservices identified by our approach that are classified in this category are considered bad.

The classification results are described in Table 3 and expressed in term of precision and recall in Table 4. Precision (resp. recall) assesses the ratio between the number of good and excellent microservices to the total number of the classified ones (resp. the number of the manually identified ones).

4.4 Answers to Research Questions

The precision values obtained based on *FMicro* are greater than 83%. This shows that a large part of the manually identified microservices are identified by our approach. The recall values obtained based on *FMicro* are also greater than 75%. This means that a large part of the microservices identified by our approach are those identified manually. Thus, we answer ***RQ1*** as follows: our approach allows obtaining an adequate decomposition of an OO application into microservices.

In addition, similarly to *FMicro*, the precision values obtained based on the partial quality function *FSem*, are between 80% and 100%.

The interpretation of the recall values for *FSem* is the same as *FMicro* while considering that the recall values are either equal to or lower than those obtained

Table 3. Microservice classification results

Application	No of excellent microservices		No of good microservices		No of bad microservices	
	FMicro	*FSem*	*FMicro*	*FSem*	*FMicro*	*FSem*
InventoryManagementSystem	1	1	17	15	3	5
FindSportMates	0	0	3	3	1	1
SpringBlog	3	2	7	8	3	3

Table 4. Precision and recall measurement

Application	Precision		Recall	
	FMicro	*FSem*	*FMicro*	*FSem*
InventoryManagementSystem	90%	80%	85,71%	76,19%
FindSportMates	100%	100%	75%	75%
SpringBlog	83,33%	83,33%	76,92%	76,92%

by relying on *FMicro*. Based on these values, the answer to ***RQ2*** is the definition of the quality function, without considering data autonomy, is adequate.

The precision and recall values obtained based on *FSem* are equal to or less than those obtained based on *FMicro*. The values that are equal are related to applications that do not manipulate many persistent data. Thus, the answer to ***RQ3*** is the evaluation of data autonomy enhance the quality of microservices.

4.5 Threats to Validity

Threats to Internal Validity: Our approach may be affected by two internal threats. Firstly, each class belongs to only one microservice. However, in some applications, some classes may participate in several functionalities. Nevertheless, this generally concerns only certain classes that the architect can duplicate. Secondly, we rely on a hierarchical clustering algorithm. This algorithm allows obtaining values of the quality function close to optimal ones.

Threats to External Validity: There are two external threats. Firstly, the quality of the OO source code can impact the identification. Secondly, the matching between the microservices obtained by our approach and those obtained manually can vary according to the granularity of the manually identified ones.

5 Conclusion

We presented, in this paper, an approach for the identification of microservices by an analysis of OO source code. This approach is based on both the evaluation of microservice quality, using a quality function, and an algorithm for grouping classes according to the value of this quality. The conducted experimentation shows the relevance of the obtained microservices using our approach compared to those identified manually. However, the results need to be consolidated by experimentations on very large applications. Moreover, inspired by existing works [2,6], we will propose an approach to package the identified microservices and deploy them on the cloud while taking into account the dynamic reconfiguration.

References

1. Adjoyan, S., Seriai, A.D., Shatnawi, A.: Service identification based on quality metrics object-oriented legacy system migration towards SOA. In: SEKE (2014)
2. Alshara, Z., Seriai, A.D., Tibermacine, C., Bouziane, H.L., Dony, C., Shatnawi, A.: Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. ACM SIGPLAN Not. **51**, 55–64. ACM (2015)
3. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15

4. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 19–33. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_2
5. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Reading (2015)
6. Bastide, G., Seriai, A., Oussalah, M.: Adapting software components by structure fragmentation. In: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 1751–1758. ACM (2006)
7. Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object-oriented system. ACM SIGSOFT Softw. Eng. Notes **20**, 259–262 (1995)
8. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: a systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 185–200. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_12
9. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint (2016)
10. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. MartinFowler.com **25** (2014)
11. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 524–531. IEEE (2017)
12. Newman, S.: Building Microservices: Designing Fine-grained Systems. O'Reilly Media, Inc., Sebastopol (2015)
13. Sharma, S.: Mastering Microservices with Java. Migrating to Cloud-Native Architectures Using Microservices: An Experience ReportPackt Publishing Ltd. (2016)
14. Sharma, S., Gonzalez, D.: Microservices: Building scalable software (2017)