Draft

Check for updates

Combining Static and Dynamic Analysis to Decompose Monolithic Application into Microservices

Khaled Sellami[1], Mohamed Aymen Saied[1(✉)], Ali Ouni[2], and Rabe Abdalkareem[3]

[1] Laval University, Quebec, QC, Canada
mohamed-aymen.saied@ift.ulaval.ca
[2] ETS Montreal, University of Quebec, Montreal, QC, Canada
[3] Carleton University, Ottawa, ON, Canada

Abstract. In order to benefit from the advantages offered by the microservices architectural design, many companies have started migrating their monolithic application to this newer design. However, due to the high cost and development time associated to this task, automated approaches need to be developed to solve these issues.

Solutions that tackle this problem can be classified based on the information available for the monolithic application which are often based on source code or runtime traces. The latter provides a more accurate representation of the interactions between the classes within the application however it often fails to cover all of the classes. On the other hand, the source code of the application is more readily available and can be used to extract additional information like semantic meaning of the classes.

The objective of this paper is to provide a hybrid solution that combines both of these approaches in order to take advantage of their strengths while covering their weaknesses. The proposed solution performs static and dynamic analysis on the monolithic application based on the available information and the user's input. Afterwards, an iterative clustering process is applied on the processed data in order to generate the microservices decomposition. We compare different strategies for combining the static and dynamic approaches and we evaluate the performance of the hybrid approach compared to each of the separate approaches on 4 monolith applications. We provide as well a comparison with state-of-the-art solutions.

Keywords: Microservices · Clustering · Legacy decomposition · Static analysis · Dynamic analysis

1 Introduction

Monolithic architectural styles implemented in the legacy applications often lead to maintainability issues as these applications evolve and as such fail to meet user demands or provide their services adequately [4]. Service Oriented Architectures (SOA) have emerged as an alternative when building new software which tries to answer the problems found in monolithic applications. The microservices architecture [1,13] builds upon the philosophy used in SOAs to utilize a

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
J. Troya et al. (Eds.): ICSOC 2022, LNCS 13740, pp. 203–218, 2022.
https://doi.org/10.1007/978-3-031-20984-0_14

Domain Driven Design (DDD) [8] to build autonomous, fine-grained and scalable components (microservices) that can function independently. A large number of developers have sought to adopt this style and migrate their legacy applications. However, this migration process proved to be costly, lengthy and complex in many cases, requiring a large amount of time and monetary investment from these developers and as such served as a barrier to improve their software [10]. Approaches that try to tackle this issue attempt at automating this part of the process by proposing the set of potential microservices which is called a decomposition. Each approach tackles this issue in a different way mostly based on the type of input it utilizes and how it analyzes it. One of the most commonly used approach relies on the information found within the run-time traces of the monolithic application [3,6,7] since it provides a more accurate view of the interactions of the components within this application. However, this approach, called Dynamic Analysis, requires the availability of enough execution traces to provide this advantage and, as such, methods that employ it often fail to cover all of the components within the application. The other most common approach uses the source code of the legacy application [11,15,16] since it is rare that this information would be unavailable for a developer that is trying to migrate his application. In addition, this analysis approach, called Static Analysis, can cover all of the components within the legacy software and include them in the decomposition.

In this research, we present a solution that merges Static Analysis and Dynamic Analysis approaches in order to complement each other by providing more robust decompositions which take advantage of the run-time traces while covering the whole application by supplementing the inference phase with the information extracted from the source code. Our solution analyzes the run-time traces and the source code independently in order to extract semantic, structural and dynamic representations of the monolithic application. Afterwards, we apply an iterative clustering approach that combines representations from different domains in order to generate a single result in a hierarchical structure that represents the microservices.

In this paper, we compare different strategies for combining the analysis approaches and we evaluate our approach in comparison with other baselines in the literature that tackle problems similar to the microservices decomposition issue. The results obtained show that our approach improved the coverage of our proposed decompositions while maintaining Structural Modularity, Conceptual Modular Quality and Inter Call Percentage metrics that are better or similar to most of the baselines.

The paper is organized as follows. In Sect. 2, we present the related work to our research. Afterwards, we showcase a formal formulation of the problem and the details of our proposed approach in Sect. 3. Then, in the 4th section, we specify and describe the empirical evaluation of our approach. Subsequently, we move on to discussing the threats to the validity of this work in Sect. 5. Finally, we provide a conclusion to the paper, and we outline our future work in Sect. 6.

2 Related Work

Recent research in the migration process from a monolithic architecture to a microservices architecture has focused mainly on the decomposition phase where given a monolithic application, an approach proposes a set of potential microservices. There has been numerous attempts to automate this task. These approaches can be categorized by how they process the monolithic application and how they analyze it.

Some solutions focused on the use of execution traces to represent the legacy systems. Mono2micro [7] associates execution traces with use cases and then analyzes them to calculate a shared similarity metric between the classes. Then, it uses a hierarchical clustering algorithm to suggest the microservices. FoSCI [6] addresses this problem by proposing a solution that relies on execution traces and a search-based algorithm to group together the classes of the monolithic application. The approach CoGCN [3] is based on a graph neural network that provides the proposed decomposition while outputting the list of outliers. This approach builds its neural network using the structural data in the source code and trains the model using the execution traces.

Most other solutions that tackle this problem rely on the source code for their analysis. hierDecomp [16] analyzes the source code in order to extract the structural and semantic information within it which is used in conjunction with a hierarchical DBSCAN algorithm variant to generate the decomposition options. Bunch [11] is a tool designed to provide an architectural-level view of a software system by decomposing it and clustering its components using search algorithms and using only the source code of the application.

Some approaches have tried to represent the monolithic applications using different sources of information. MEM [9], for example, relies on the source code and the version control history of the application to generate a graph. It proposes its microservices by applying a clustering algorithm on this graph. ServiceCutter [5] takes as input a JSON format of the design documents of the monolithic application. Using this input, ServiceCutter generates scores for 16 coupling criteria and generates a weighted graph. The developers can use this graph to generate a service oriented architecture.

3 Proposed Approach

In this section, we present the details of our solution. We start by defining the problem we are trying to solve. Afterwards, we showcase an overview of the proposed approach. Then, we explain in detail the different components used in this approach.

3.1 Problem Formulation

Given a legacy monolithic application, our approach needs to generate a set of candidate microservices which is called in this case a decomposition. This task

is achieved by analyzing the source code and execution traces. Even though this solution can be applied on each one of these inputs individually, we will assume that both types of information are available for the rest of the paper.

The legacy application is represented as a set of classes $C = \{c_1, c_2, ..., c_N\}$ where c_i is the class's id and N is the total number of classes. In addition, given that dynamic analysis rarely covers all of the classes within the code base, we define $C_d = \{c'_1, c'_2, ..., c'_{N_d}\}$ as the set of classes mentioned within the execution traces where $c'_i \in C$ and $N_d \leq N$.

The result of our approach is a hierarchical representation of the suggested decomposition. It is defined as a list of layers, each representing a level of the hierarchy. The ith layer is defined as $L_i = \{M_{i,1}, M_{i,2}, ..., M_{i,N_i}\}$ where $M_{i,j} = \{c_{i,j,1}, c_{i,j,,2}, ..., c_{i,j,N_{i,j}}\}$ is a microservice containing $N_{i,j}$ classes and $c_{i,j,k} \in C$. If a microservice contains only one class, that class is defined as an outlier. In addition, for each microservice $M_{i,j}$ in the ith layer, there exists a microservice $M_{i+1,j'}$ in the $(i + 1)^{th}$ layer where $M_{i,j} \subseteq M_{i+1,j'}$.

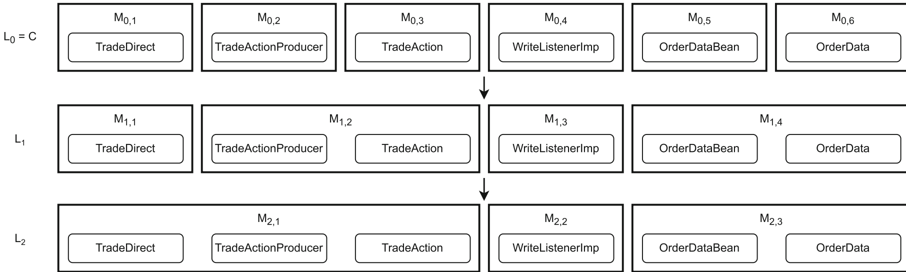

Fig. 1. An example showcasing the result of a microservice decomposition.

Figure 1 showcases an example of a decomposition results for a small subset of classes within an open-source monolithic Java application called *DayTrader*[1]. The initial layer is defined as a set of microservices each having exactly one class. The second layer contains 4 microservices since the couple of classes *TradeAction* and *TradeActionProducer* as well as *OrderData* and *OrderDataBean* have been merged into a single microservice each. Since the microservices $M_{1,1}$ and $M_{1,3}$ have only the classes *TradeDirect* and *WtiteListenerImp* respectively, both of these classes are categorized as outliers within this layer. For the final layer, the microservices $M_{1,1}$ and $M_{1,2}$ have merged to create the 3-class microservice $M_{2,1}$. As such, the suggested decomposition contains 2 microservices and *WriteListenerImp* as the only outlier.

Having defined the input and output of our solution, the following subsection explains the details of our approach as well as the theoretical reasoning behind it.

[1] https://github.com/WASdev/sample.daytrader7.

3.2 Approach Overview

Our approach takes as input the source code and execution traces of a given monolithic application. Afterwards, three separate and distinct analysis approaches are executed on this input in order to generate a dataset for each approach. The three datasets are then fed to the clustering component which combines them in order to output the decomposition layers. Nonetheless, any combination of the analysis approaches is possible including having a single one.

The Fig. 2 showcases the different steps taken in order to generate a decomposition for a given monolithic application. The smaller rectangles within the figure represent the task done by our solution while the ellipses represent inputs and outputs.

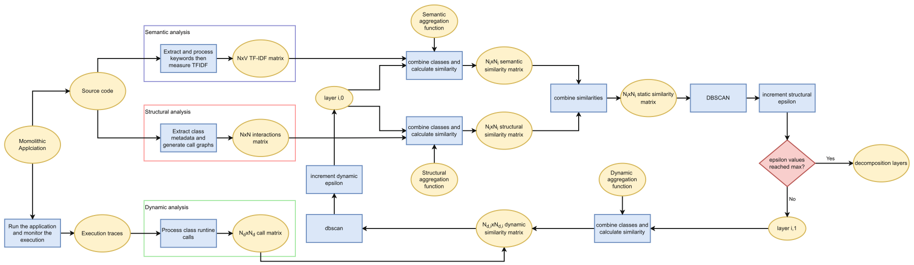

Fig. 2. An overview of the process used to output the microservices decomposition.

3.3 Extracting the Datasets for Each Approach

Dynamic Calls Matrix. This phase requires as input a list of execution traces recording the dynamic interactions of the classes. These traces represent the execution logs. Each trace should represent a call path from the first class until the last called class. Branches in the call path create another trace. For example, if during an execution, *TradeActionProducer* called *TradeAction* which then called *TradeDirect* this would create the first trace: [*TradeActionProducer*, *TradeAction*, *TradeDirect*]. If *TradeDirect* finished its task and returned, and afterwards *TradeAction* called *OrderData*, we would create a second trace: [*TradeActionProducer*, *TradeAction*, *OrderData*]. All circular dependencies within the traces and all duplicates are removed in a pre-processing step. Using these traces, we generate the dynamic calls matrix. We define the dynamic calls matrix M_{dyn} as a N_dxN_d matrix where each cell is equal to the sum of direct calls and indirect calls between every couple of classes within the execution traces. For example, given the following traces: [*TradeActionProducer*, *TradeAction*, *TradeDirect*] and [*TradeActionProducer*, *TradeDirect*] and the order of classes is [*TradeActionProducer*, *TradeAction*, *TradeDirect*], the call matrix would be equal to : $\begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

Structural Interactions Matrix. We define an interaction between a class A and a class B when within class A, class B was declared, used as a type for a method's parameter, inherited or had one of its methods called. In addition, all classes acquire the interactions of the class they inherit from.

As such, we start by extracting the metadata within the source code. Any static analysis tool that can analyze the Abstract Syntax Trees of the application's programming language can be used to extract this information. Afterwards, for each couple of classes, we measure the number of interactions between them in order to create the structural interactions matrix M_{str} which is a NxN matrix.

Term Frequency - Inverse Document Frequency (TF-IDF) Matrix. For each of the N classes within the source code, we extract the text used in the class' definition. The text includes the class' name, the comments, the members' names, the methods' names, the parameters' names and the variables' names within its methods. Afterwards, for each word in the text, we apply camelcase case splitting which separates the input string into multiple words based on the camelcase naming convention. For example, *CamelCase* will be split into *Camel* and *Case*. Then, we filter out stopwords. Finally, we apply a stemming process in order to facilitate the detection of similar words. After this pre-processing step, we acquire a vector of words for each class which is used, in conjunction with the vocabulary V to measure the TF-IDF values and obtain the TF-IDF matrix M_{sem}. The final result would be a N xD_V matrix where D_V represents the number of words in the vocabulary.

3.4 The Hybrid Clustering Process

The objective of this task is to combine the different matrices generated in the previous task in order to provide a better decomposition than each of the approaches separately. Both structural and semantic analysis can utilize similarity functions that generate $N \times N$ matrices whose values are in the range *[0,1]* where N refers to the total number of classes within the monolithic application. For this reason, an intuitive and simple solution would be to calculate the weighted sum of structural and semantic similarity matrices using a weight value called alpha in the range *[0,1]*. For the rest of the paper, we will call this matrix the static analysis matrix since it's based on a couple of approaches that employ static analysis.

On the other hand, the dynamic calls matrix can't be used to generate a NxN matrix since it lacks information regarding some of the classes. As such, a simple weighted sum is not sufficient. In this case, we use a clustering strategy that combines 2 datasets from different domains in order to generate a single clustering result introduced in [14]. This approach builds upon a modified DBSCAN algorithm [12,16].

This algorithm, which we call hierarchical-DBSCAN, executes DBSCAN in multiple iterations and slowly increments the epsilon hyper-parameter in order

to loosen the restriction on the condition for grouping together the classes until a maximum epsilon value, defined by the user, is reached. Each iteration takes as input additionally the clustering of the previous iteration. As such, the final result is a list of layers describing the hierarchy of the clusters since each cluster with a layer contains at least one of the clusters of the previous layer similarly to the example shown in Fig. 1.

Combination Strategy. The algorithm introduced in [14] proposed two different strategies to combine the datasets. The first strategy involves running the hierarchical-DBSCAN processes separately and in a sequential manner.

As shown in Fig. 3, we start with one of the datasets, which in our case is the dynamic call matrix and we execute all of the iterations of the hierarchical clustering algorithm. At each iteration, we take as input the previous iteration's result and the original dataset. Then, for each cluster in the previous layer, we generate a new sample that represents the cluster depending on an aggregation function. Afterwards, we calculate a similarity matrix based on the newly created samples. Using the similarity matrix, we run the DBSCAN algorithm in order to acquire the new clusters. After incrementing the epsilon parameter, we verify if it exceeds a maximum threshold called Max epsilon and that is defined by the user. If it does not, we feed the clustering result to the next iteration. Otherwise, we feed it as input into the second phase which applies the same process on the second dataset, its corresponding aggregation function and its Max epsilon hyper-parameter. Finally, when the second epsilon reaches its maximum, the acquired clustering layers are returned as the output of the algorithm.

The Fig. 4 showcases the second strategy. In this case, we alternate between the datasets. We start by running an iteration for the first dataset. Afterwards, we update the first epsilon value and we feed the result to an iteration of the second dataset. Similarly, we update the second epsilon value and use the result as the input of the second iteration of the first dataset. We keep alternating between both datasets until both epsilon values have reached their respective maximum values. Finally, we output the clustering layers.

Given the assumption that dynamic analysis data are a better representation of the application at the cost of a lower class coverage, we always start the clustering process with the dynamic call matrix as the first dataset.

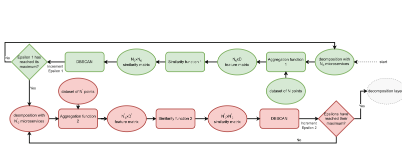

Fig. 3. A showcase of the sequential strategy.

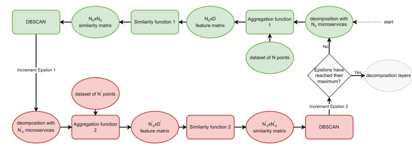

Fig. 4. A showcase of the alternating strategy.

Aggregation Functions. During each iteration and for each different type of analysis, we take as input the previous clusters and the original dataset. We define a function capable of aggregating each cluster into a single point. The newly generated vectors replace the vectors of the clusters' components within the dataset. The resulting dataset is then used in the next steps of the current iteration

For semantic analysis, each cluster is transformed into a normalized vector representing the mean of the TF-IDF vectors of its classes. Given a cluster C, we generate the new vector as:

$$\vec{c_C} = \frac{\sum_{c_i \in C} \vec{M_{sem}[c_i]}}{|C|} \tag{1}$$

where $\vec{M_{sem}[c_i]}$ is the vector encoding the class i in the TF-IDF Matrix M_{sem}

As for both structural and dynamic analysis, we use the same aggregation function which measures the sum of the vectors representing its classes. Given a cluster C and the label a in $\{dyn, str\}$, we generate the new vector as:

$$\vec{c_C} = \sum_{c_i \in C} \vec{M_a[c_i]} \tag{2}$$

4 Evaluation

In this section, we conduct experiments in order to evaluate the performance of our approach in identifying the optimal decomposition.

4.1 Research Questions

We developed our experimental setups in order to answer the following research questions:

- **Q1:** What is the best approach for combining different representations and interpretations of the monolithic application?
- **Q2:** How does our approach perform when compared to state-of-the-art microservices decomposition baselines?

4.2 Experimental Setup

Evaluation Metrics. In order to properly evaluate our solution and compare it with other approaches, we need to define metrics that can quantify the quality of the generated microservices. However, since we are dealing with a problem that does not contain true values we can compare with, we will need to evaluate the quality of the decomposition based on defined criteria that theoretically represent an acceptable microservices architecture [10]. As such, for this evaluation, we will compare the proposed decompositions based on how much the decomposition respects the Domain Driven Design (DDD) philosophy [8], how coherent the

microservices are, how much coupling exists between them and the granularity of the microservices

For these reasons, we selected the following evaluation metrics from the literature that encode in different ways the selected criteria:

- **Structural Modularity (SM)**: [6] is an evaluation metric that can be associated with both the cohesion and coupling criteria since it defines a way to quantify the structural coherence of the microservices as well as the coupling between then combines them into a single metric. It is defined as follows:

$$SM = \frac{1}{K}\sum_{i=1}^{K}\frac{\mu_i}{m_i^2} - \frac{1}{(K(K-1))/2}\sum_{i\neq j}^{K}\frac{\sigma_{i,j}}{2\,m_i\,m_j} \tag{3}$$

  Where K is the number of the extracted microservices, μ_i is the number of unique calls between the classes in microservice i, m_i is the number of classes in microservice i and $\sigma_{i,j}$ is the number of unique calls between classes of microservice i and classes of microservice j. Decompositions with higher cohesiveness and lower coupling result in higher SM values and as such reflect a higher structural quality.
- **Conceptual Modular Quality(CMQ)**: [6], quantifies the conceptual quality of the decomposition. The cohesion and coupling components within this metric are based on the common textual terms between the classes. As such, this metric evaluates how focused the contexts represented by the microservices are. Thus, it can be categorized as a metric for evaluating the DDD aspects.

$$CMQ = \frac{1}{K}\sum_{i=1}^{K}\frac{\mu'_i}{m_i^2} - \frac{1}{(K(K-1))/2}\sum_{i\neq j}^{K}\frac{\sigma'_{i,j}}{2\,m_i\,m_j} \tag{4}$$

  Where K is the number of the extracted microservices, μ'_i is the number of common terms between the classes in microservice i, m_i is the number of classes in microservice i and $\sigma'_{i,j}$ is the number of common terms between classes of microservice i and classes of microservice j. Higher CMQ values reflect better decompositions.
- **Non-Extreme Distribution (NED)**: [3] This metric corresponds to the granularity criteria and introduces a way to quantify this aspect by measuring the percentage of classes with extremely small or extremely large microservices. It is defined in detail in the following equation:

$$NED = 1 - \frac{|\{m_i\ ;\ 5 < |m_i| < 20, i \in [1, K]\}|}{K} \tag{5}$$

  Where K is the number of the extracted microservices and $|m_i|$ is the size of microservice m_i. In our evaluation, we selected the values 5 and 20 as the thresholds for the definition of extreme sizes for all sample applications in order to be consistent with the literature [2,3,7]. Having high NED often corresponds to worse results.

- **Inter Call Percentage (ICP)**: [7] is based on the percentage of static calls between two microservices. This metric quantifies the dependencies between the microservices and as such can represent the coupling criteria.

$$ICP = \frac{\sum_{i=1,j=1,i\neq j}^{K} \sum_{c_k \in M_i} \sum_{c_l \in M_j} (log(calls(c_k, c_l)) + 1))}{\sum_{i=1,j=1}^{K} \sum_{c_k \in M_i} \sum_{c_l \in M_j} (log(calls(c_k, c_l)) + 1)} \quad (6)$$

  Where K is the number of microservices, M_i is the set of classes in microservice i, $calls(c_k, c_l)$ is the number of calls from class c_k to class c_l. Lower values of ICP correspond to fewer interactions and as such lower coupling and a better decomposition.
- **Coverage (COV)**: is simply defined as the percentage of classes from the monolithic application that were included in the proposed decomposition. For our approach, we won't consider outlier classes as part of the proposed decomposition. If we measure this metric for the decomposition example shown if Fig. 1 which has 5 classes and detected 1 outlier, the result would be equal to 0.8. On the other hand, if the used approach is only based on run-time execution trace analysis and only 3 classes were detected, the result for this approach would 0.6.

Evaluation Applications. We selected 4 monolithic Open-source Java applications that we evaluate our approach on. The selected applications have varying scales in order to evaluate how scalable our approach is. The metadata of these applications are described in the Table 1 where we specify the number of classes detected using static analysis (SA) and dynamic analysis (DA) separately and the number of unique interactions found using static analysis.

Table 1. Monolithic applications metadata.

Project	Version	SLOC	# of SA classes	# of DA classes	# of unique interactions
Plants	1.0	7,347	40	20	123
JPetStore	1.0	3,341	73	37	209
AcmeAir	1.2	8,899	86	23	242
DayTrader(see footnote 1)	1.4	18,224	118	73	378

[1] https://github.com/WASdev/sample.mono-to-ms.pbw-monolith.
[2] https://github.com/KimJongSung/jPetStore.
[3] https://github.com/acmeair/acmeair.

Experimental Process. For each research question, we propose different alternatives that we compare their results. However, hyper-parameter choices can significantly impact the quality of the output. As such, we applied a grid-search like approach where we select intervals of possible values for each hyper-parameter that is not under evaluation and then we generate the decompositions for each

hyper-parameter combination and we measure their evaluation metrics. Afterwards, we filter out the decompositions that have a NED score equal to 1. Since NED is calculated by the percentage of microservices with extreme sizes, having a NED score equal to 1 signifies that all the microservices within this decomposition can be considered invalid and as such this solution should be excluded. Additionally, we exclude the decompositions that have a coverage lower than a defined threshold. In this process, we used 0.5 as the threshold.

4.3 Experimental Setup and Results for RQ1

In this research question, we evaluate which combination strategy as described in the section Combination strategy performs better. Therefore, we start by comparing the performance of the sequential strategy and the alternating strategy.

After applying the experimental process and excluding the extreme cases, we evaluate the influence of the chosen strategy independently from the hyper-parameters based on the analysis of over 40000 potential decompositions. The Table 2 shows the median result for each evaluation metric, sample application and strategy.

Table 2. Comparison of median evaluation results for approach combination strategies.

	SM ↗		CMQ ↗		ICP ↘		NED ↘		COV ↗	
	Alternating	Sequential	Alternating	Sequential	Alternating	Sequential	Alternating	Sequential	Alternating	Sequential
Plants	0.4037	0.4042	0.0385	0.0246	0.1776	0.1752	0.3077	0.3478	0.675	0.65
JPetStore	0.0767	0.0789	0.1647	0.1539	0.3378	0.4641	0.5968	0.5082	0.863	0.8493
AcmeAir	0.093	0.1031	0.3127	0.2757	0.3885	0.5799	0.7229	0.6125	0.8652	0.7753
DayTrader	0.2219	0.227	0.2047	0.1991	0.2425	0.347	0.7103	0.6848	0.8305	0.7627

As we can observe in the table, both methods had very close median results for the metric SM with the largest difference being around 0.004 for the project AcmeAir. However, we can see that using the alternating strategy achieved higher results for all projects. As for ICP, the alternating strategy managed to lower its values and achieve a worse but very close median score compared to the sequential strategy. On the other hand, when comparing the scores for NED, we can see that the alternating had more extreme microservices in all projects except for Plants. Finally, the coverage it achieved was better in all applications.

We hypothesize that the increased performance observed in this case is due to the feedback loop between the clustering processes that exists in the alternating strategy compared to the sequential approach. In the first case, the results of the dynamic analysis clustering process feed into the static analysis clustering process at each iteration which should improve the quality of this process and vice versa. As for the sequential strategy, the results of the dynamic analysis clustering process are only used as the input for the first iteration of the static analysis clustering process.

For the following experiments, we will exclusively use the alternating strategy.

> Using the alternating strategy when combining the static and dynamic analysis results generated decompositions that had better metrics, in general, than those achieved by the sequential strategy decompositions.

4.4 Experimental Setup and Results for RQ2

In order to answer this research question, we selected the six approaches that tackle the monolithic to microservices decomposition problem or a similar problem using different methods and views of the monolithic applications. These approaches are **Bunch** [11], **CoGCN** [3], **Hierarchical DBSCAN (HierDec)** [16], **FoSCI** [6], **MEM** [9] and **Mono2micro (M2M)** [7].

For each one of the baselines we compare with as well as our approach (HyDec), we use different ranges of hyper-parameters in order to generate multiple decompositions. Then we calculate all five of the evaluation metrics. Similarly to the previous research questions, we eliminate all decompositions that have a NED score equal to 1.

Figure 5 showcases the results of each baseline for each metric and each sample application in boxplot figures. Our solution is highlighted in red.

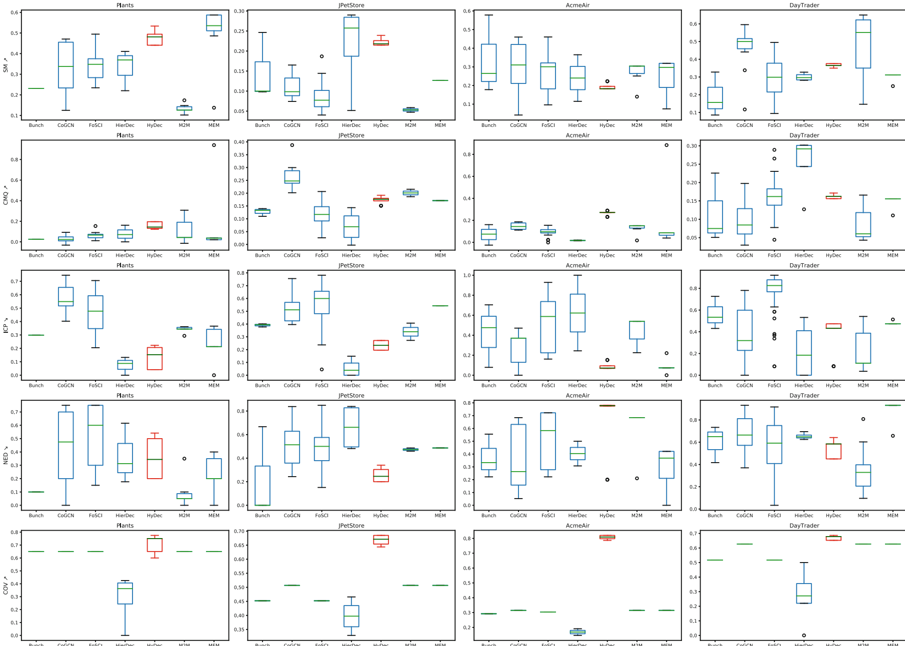

Fig. 5. Boxplots of the evaluation results for each baseline. (Color figure online)

For the sample application Plants, we can observe in the Figure that our approach achieved the highest CMQ median score while managing to have the second highest SM median score and second best ICP score. Only MEM and HierDec managed to have a better score than our approach respectively in SM and ICP. As for NED, our solution had a better score than MEM, FoSCI and CoGCN while M2M achieved the lowest NED. Finally, Our approach had the highest coverage while HierDec had the lowest.

As for JPetStore, our approach managed to achieve the second-best score in both SM and ICP in which HierDec had the best score. However, our approach had significantly better NED and coverage score than the rest of the baselines with only Bunch as an exception for the NED metric. Although HyDec did not reach the best score for CMQ like in the case of Plants, its score managed nonetheless to be the third best and is very close to M2M's score.

When comparing our approach with the rest of the baselines in the AcmeAir project, we can see that it achieved much better coverage than the rest where the median is at least twice as much as the second highest coverage. In addition, it had the highest CMQ and a similar median score to the highest result in ICP which was acquired by MEM. However, these scores came at the cost of lower SM values and higher NED values.

Finally, by comparing the results generated for the application DayTrader using our approach to those created by the other baselines, we can see that HyDec had the highest coverage, the second highest CMQ score, the second-best NED score and the third-highest SM score. As for ICP, our approach managed to have a better score than 3 out of the 7 baselines.

HyDec had the best median COV in all of the sample applications since our approach does not rely too heavily on the run-time execution traces but instead combines it with the source in order to improve the results while having enough information to place as many classes as possible into their adequate microservices. In addition, HyDec managed to be within the 3 best approaches for all sample applications for the metrics SM, CMQ and ICP with the exception of a couple of cases: SM for AcmeAir and ICP for DayTrader. These results showcase that even with the higher coverage, which serves as a disadvantage when calculating these metrics, our approach still managed to improve over the baselines for some cases and remain competitive for the rest. As for NED, the results varied from one application to another. For example, even though HyDec had a significantly higher coverage than the baselines, it did not negatively affect the NED score unlike what happened with AcmeAir. As for the other applications, HyDec's NED score was close to the average of the baselines.

Our approach, HyDec, managed to increase the coverage of the decomposition and to achieve better conceptual and static cohesion and coupling than the other baselines in most cases.

5 Threats to Validity

For internal threats to validity, the biggest threat lies within the selection evaluation metrics and the hyper-parameters for our approach. For the former, we tried to use five metrics that differ in the criteria that they represent and that use different inputs, except for the proposed decomposition, to calculate. As for the latter, we tried to mitigate this threat by varying these hyper-parameters in order to generate multiple decompositions for the comparison. Particularly for the comparison with the baselines, we applied the same process and the same conditions on all of the approaches. The implementation of the approaches could be a threat to the validity of this research as well. We attempted to mitigate this issue by extensively testing the code and verifying the obtained results.

In this paper, we evaluated our approach on only four monolithic applications. Although we tried to select a set of applications that have varying numbers of classes and interactions, it would be beneficial to our research to evaluate its performance on additional sample monolithic applications. Our approach uses the classes of the monolithic application as the granularity level of its representation. There is a debate within the literature on which granularity level would be more suitable for the decomposition task [6]. In our case, we decided on the class level since this research focused mainly on Object-Oriented Languages for which the classes represent a core concept when coding. Having a more fine-grained level, like for example at the procedural level, can lead to more coupling issues and as such more refactoring would be required.

6 Conclusion and Future Work

We presented a microservices decomposition solution that takes as input the source code of a monolithic application as well as run-time traces of its execution. The proposed approach analyzes each of the sources individually extracting semantic and structural information of the classes within the monolithic application from the source code and dynamic interactions between the classes from the execution traces. Then, an iterative clustering process starts which groups together the classes based on the current analysis type, the results of the previous layer and the current constraints. The final result is a hierarchical view of the proposed microservices. The evaluation results showcase that this approach improves over individual applications of each analysis approach and a comparison with state-of-the-art approaches shows that our solution managed to surpass the coverage of the rest of the baselines while providing decompositions that have competitive structural and conceptual cohesion and coupling.

In the future, we would like to work on improving the analysis phase of our approach, and particularly the semantic analysis approach in order to extract more accurate information from the source code of the monolithic applications. We would like to investigate as well if we can combine information extracted from

other sources like the version control history or the documentation. Finally, it would be interesting to study the impact of prioritizing the domain relationship between the classes over the structural and dynamic interactions and find a way to evaluate whether these solutions would be more beneficial.

References

1. Benomar, O., Abdeen, H., Sahraoui, H., Poulin, P., Saied, M.A.: Detection of software evolution phases based on development activities. In: 2015 IEEE 23rd International Conference on Program Comprehension (2016)
2. Bittencourt, R.A., Guerrero, D.D.S.: Comparison of graph clustering algorithms for recovering software architecture module views. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR (2009)
3. Desai, U., Bandyopadhyay, S., Tamilselvam, S.: Graph neural network to dilute outliers for refactoring monolith application (2021)
4. Fritzsch, J., Bogner, J., Wagner, S., Zimmermann, A.: Microservices migration in industry: Intentions, strategies, and challenges. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2019)
5. 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
6. Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R., Zheng, Q.: Service candidate identification from monolithic systems based on execution traces. IEEE Trans. Softw. Eng. **47**(5), 987–1007 (2019)
7. Kalia, A.K., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., Banerjee, D.: Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. In: ESEC/FSE 2021. Association for Computing Machinery Inc (2021)
8. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term (2017)
9. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: Proceedings - ICWS 2017. Institute of Electrical and Electronics Engineers Inc (2017)
10. Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S.T., Dustdar, S.: Microservices: migration of a mission critical system. IEEE Trans. Serv. Comput. **14**(5), 1464–1477
11. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. **32**(3), 193–208 (2006)
12. Saied, M.A., Ouni, A., Sahraoui, H., Kula, R.G., Inoue, K., Lo, D.: Improving reusability of software libraries through usage pattern mining. J. Syst. Softw. **145**, 164–179 (2018)
13. Saied, M.A., Raelijohn, E., Batot, E., Famelis, M., Sahraoui, H.: Towards assisting developers in API usage by automated recovery of complex temporal patterns. Inf. Softw. Technol. **119**, 106213 (2020)
14. Saied, M.A., Sahraoui, H.: A cooperative approach for combining client-based and library-based API usage pattern mining. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC) (2016)

15. Sellami, K., Ouni, A., Saied, M.A., Bouktif, S., Mkaouer, M.W.: Improving microservices extraction using evolutionary search. Inf. Softw. Technol. **151**, 106996 (2022)
16. Sellami, K., Saied, M.A., Ouni, A.: A hierarchical dbscan method for extracting microservices from monolithic applications. In: The International Conference on Evaluation and Assessment in Software Engineering 2022. Association for Computing Machinery (2022)