From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts

Luís Nunes, Nuno Santos, and António Rito Silva(✉)

INESC-ID/Department of Computer Science and Engineering,
Instituto Superior Técnico, Av. Rovisco Pais 1, 1049-001 Lisbon, Portugal
{luis.a.nunes,nuno.v.santos,rito.silva}@tecnico.ulisboa.pt

Abstract. Microservices have become the software architecture of choice for business applications. Initially originated at Netflix and Amazon, they result from the need to partition, both, software development teams and executing components, to, respectively, foster agile development and horizontal scalability. Currently, there is a large number of monolith applications that are being migrated to a microservices architecture. This article proposes the identification of business applications transactional contexts for the design of microservices. Therefore, the emphasis is to drive the aggregation of domain entities by the transactional contexts where they are executed, instead of by their domain structural inter-relationships. Additionally, we propose a complete workflow for the identification of microservices together with a set of tools that assist the developers on this process. The comparison of our approach with another software architecture tool and with an expert decomposition in two case studies revealed high precision values, which reflects that accurate service candidates are produced, while providing visualization perspectives facilitates the analysis of the impact of the decomposition on the application business logic.

Keywords: Monolith applications · Microservices architecture · Architectural migration · Transactional logic decomposition

1 Introduction

Microservices architecture [22] is increasingly being adopted as the software architecture of business applications. Initially originated at Netflix and Amazon, they result from the need to partition, both, software development teams and executing components. The former promotes the application of software agile approaches, due to smaller loosely dependent teams associated to partitions of the domain model, while the later improves the system horizontal scalability, due to the ability to have different levels of scalability for each execution context. On the other hand, a large number of existing applications are implemented using

T. Bures et al. (Eds.): ECSA 2019, LNCS 11681, pp. 37–52, 2019.
https://doi.org/10.1007/978-3-030-29983-5_3

the monolith architecture, where a single database is shared by all the system functionalities.

A survey done with experts identifies *Wrong Cut*, when microservices are split in the basis of technical layers instead of business capabilities, as one of the two worst bad practices when designing microservices [21]. Therefore, several approaches [12] are being proposed on how to migrate monolith systems to a microservices architecture. Most of these approaches are driven by the identification of structural modules, which have high cohesion and low coupling, in the monolith domain model. However, they do not consider the need to change the application business logic when migrating a monolith to a microservices architecture. This problem is identified in [8] as the *Forgetting about the CAP Theorem* migration smell, which states that there is a trade-off between consistency and availability [6].

We propose the identification of business applications transactional contexts for the design of microservices. Therefore, the emphasis is to drive the aggregation of domain entities by the transactional contexts where they are executed, instead of by their structural domain inter-relationships. This equips software architects to reason about the impacts of the migration on the overall system business logic, due to the relaxing of consistency. Additionally, we define a complete workflow for the identification of microservices together with a set of tools that assist the architects in this process, and apply concepts and techniques such as code analysis and graph clustering.

The comparison of our approach with another software architecture tool and an expert decomposition of the case studies resulted in high precision values, which reflects that accurate service candidates are produced.

The subsequent sections are going to be summarized as follows. Section 2 presents the concepts behind architectural migrations to microservices, Sect. 3 describes the proposed solution, where Sect. 4 evaluates the result of applying the automatic decomposition to a monolith application. Finally, Sect. 5 presents the related work and Sect. 6 the conclusions.

2 Concepts

The migration of a monolith to a microservices architecture comprises three phases: data collection, which collects information about the system that is intended to migrate to microservices architecture; microservices identification, where one, or several, criteria are applied to the collected data to identify microservices candidates; and visualization, which provides a visual representation of the identified microservices, and their relationships, according to different perspectives.

The approaches differ on which technique they use to collect the data, either manual or automatically, the type of data collected, e.g. a call graph or the execution log of the application, and if they are source code based or model-based, which in the latter case the data collection corresponds, actually, to a modeling activity followed by the extraction of information from the model.

The automatic collection of data is based on techniques like, static analysis [20] and dynamic analysis [4], which provide different types of information. Dynamic code analysis can provide richer information, for instance the number of times an entity is accessed, but it is more difficult to obtain, because it is necessary to execute the system according to the adequate execution profiles. For instance, the number of times an entity is accessed depends on the particular usage of the system, which may even be periodic.

On the other hand, the type of collected information is strongly related to how each of the authors characterize a microservice and what they consider as the relevant qualities of a microservices system. For instance, some approaches use the log of the commits in a version control system repository, because they emphasize the team work separation quality of microservices architectures, while other approaches collect the number of invocations between domain entities, because they intend to reduce the communication cost between microservices in order to achieve good performance.

In what concerns the model-based approaches, they define high level representations of the system, for instance use case models and business process models, to represent the information considered necessary for the identification of microservices, arguing that the monolith may have an initial poor design and it is necessary to do some reverse engineering activities. Additionally, these approaches may be applied to the development of a microservices system from scratch. However, the possible mismatch between the source code and its model representation may hinder the microservices extraction to be done by the developer, once the microservices are finally identified by architect. Actually, according to a recent survey [11], industry experts rely on low-level sources of information, like the source code and test suites, which means that even if a model-based approach is followed, the existence of tools that analyze the source code cannot be completely dismissed.

In the microservices identification phase a metric is defined over the collected data. By using this metric a similarity measure between the system elements is calculated, such that a clustering algorithm can be applied to aggregate the monolith entities, maximizing the intra-cluster similarity and minimizing the inter-cluster similarity, where each cluster becomes a microservice candidate. Some of the approaches do not even suggest the application of a clustering algorithm but foster the identification of the microservices by the human observation of a graph, where the similarities between the monolith elements are visually represented.

Obviously, there is a close relationship between the metric and the type of data collected, for instance, if the data is about the invocations between microservices, then the metric gives a high similarity value between two monolith elements, if they have a high number of mutual invocations, such that they can be part of the same cluster.

The visualization phase uses the collected data and, together with the metric of the previous phase, presents a graph that can be analyzed according to different perspectives. For instance, it may be possible to visualize information

associated with edges between cluster nodes, for instance, the number of invocations, such that the architect can reason on the impact of these dependencies on the final microservices architecture.

Sixteen microservices coupling criteria are presented in [13]. They extract the coupling information from models and create an indirect, weighted, graph to generate clusters, using two different algorithms that define priorities for each one of the criteria. Finally, the result clusters are visualized. Although they provide the most extensive description of coupling criteria, by being based on models, they require, for some of the criteria, that part of the identification is already done. For instance, for the consistency criticality criteria it is necessary to provide information about the consistency level between the monolith elements, high, eventual, and weak. However, the identification of this information already assumes that the monolith is somehow divided and the impact of the migration in the business logic already identified, because in a monolith the consistency between its elements is always high, due to ACID transactions and their strict consistency.

3 Decomposition by Transactional Contexts

The main objective of this paper is to present a set of tools that support software architects on the process of migrating from a monolithic to a microservices architecture. Our solution relies on the identification of transactional contexts where a business transaction is divided into several transactional contexts.

We assume a software architecture for the monolith that applies the Model-View-Controller (MVC) architectural style, where the business transactions are represented by the controllers. In the monolith, the execution of a controller corresponds to the transactional execution of a request. Therefore, the monolith was designed considering the sequences of these requests, where each one of them is implemented by an ACID transaction and strict consistency. In order to reduce the impact of the migration on the system design we intend to group the domain entities accessed by a controller inside the same microservice, avoiding the introduction of relaxed consistency to the business functionality. Therefore, ideally, a controller would be implemented by a single microservice encapsulating its own database. However, there are domain entities that are accessed by several controllers. Our metric gives lower values to domain entities that are accessed the same controllers, such that they can be located in the same cluster.

Although the tools were developed for an implementation of the monolith in Spring-Boot Java and using the FénixFramework [7] object-relational mapper (ORM), the overall approached can be applied to any monolith that follows the MVC style. The FénixFramework generates a set of classes that represent the domain entities, contain the persistent information, and correspond to the data access layer. Therefore, in the first phase we do a static analysis to the monolith source code to collect, for each controller, which classes generated by

the FénixFramework are accessed. This static analysis captures the controllers call graphs using the java-callgraph[1] tool.

The metric is then implement as a similarity measure using the following formula, which returns higher values for pairs of domain entities that are accessed by the same controllers:

$$W_{E1E2} = \frac{N_{Ctrl}(E1E2)}{N_{Ctrl}(E1)} \tag{1}$$

Where, given two domain entities, $E1$ and $E2$, the weight from entity $E1$ to entity $E2$ is the quotient between the number of controllers for which their invocation tree has both, $E1$ and $E2$, as nodes ($N_{Ctrl}(E1E2)$) and the total number of controllers for which their invocation tree has $E1$ as a node ($N_{Ctrl}(E1)$). When applying this measure to a clustering algorithm, in an ideal decomposition, the entities in the same cluster are accessed by the same controllers. The domain entities are clustered using a hierarchical clustering algorithm implemented by the Scipy[2] Python library which generates a dendrogram. Finally, a user interface is used where the software architect can experiment with several cuts in the dendrogram to generate different sets of clusters. After a cut in the dendrogram is done, we support additional experimentation by allowing the architect to rename, merge and split clusters, as well as move an entity between clusters.

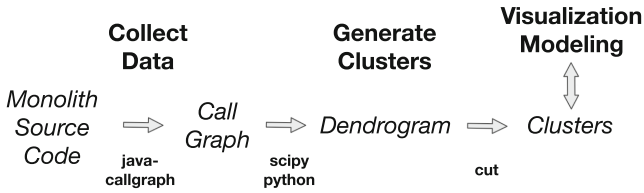


Fig. 1. Data flow schema of the tools to be developed.

The overview of the process behind the examination of the monolithic application can be seen in Fig. 1, and has the following workflow:

1. **Collect Data:** The architect uses a static code analyser implemented using the java-callgraph to generate the text call-graph.
2. **Generate Clusters:** The architect interacts with the web application to generate the dendrogram from the call-graph, using a hierarchical clustering algorithm. Afterwards, cuts the dendrogram, given a value for the maximum distance between domain entities inside each cluster, generating a set of clusters.

[1] https://github.com/gousiosg/java-callgraph.
[2] https://www.scipy.org/.

3. **Visualization:** The architect visualizes the generated information according to three views: clusters of entities and how they are accessed by controllers; the accesses pattern of controllers on clusters; the impact of domain entities data on controllers executing in other clusters.
4. **Modeling:** The architect can manipulate each one of the views, which supports informed experimentation because the tool recalculates the weights whenever a change is done.

4 Evaluation and Discussion

The approach was applied to two monolith web applications, LdoD[3] and Blended Workflow[4], but for the sake of space, only the results of the LdoD analysis are presented in the article. The analysis of the Blended Workflow provided similar insights.

4.1 LdoD

The LdoD archive[5] is a collaborative digital archive that contains the Book of Disquiet, originally written by Portuguese poet, Fernando Pessoa. LdoD monolith contains 152 controllers and 55 domain entities, being that 37 of the controllers do not make contact with the domain (24% of the systems controllers).

After applying the java-callgraph tool to collect data, and the hierarchical clustering algorithm to generate the dendrogram, we have analyzed the result according to different cuts of the dendrogram, which produce distinct cluster configurations, candidate microservices.

4.2 Metric Evaluation

As supported by the evaluation of other approaches for software architecture recovery [3,17], an internal and external assessment of the clusters is made.

Internal Evaluation. To perform an intrinsic evaluation of the clustering results for our applications, we have done an ad hoc analysis with metrics proposed by us, except for the silhouette score. These metrics allows us to compare the quality of the clustering resulting from the different cuts.

1. **Number of Singleton Clusters (NSC)**, being that having more than 2 singleton clusters is considered negative. Considering a final microservice architecture with clear functional boundaries established, it is likely that there are not two services in which their content is a single domain entity.

[3] https://github.com/socialsoftware/edition.
[4] https://github.com/socialsoftware/blended-workflow.
[5] https://ldod.uc.pt.

2. **Maximum Cluster Size (MCS)**, should not be bigger than half of the size of the system. Even with a cluster size inside this range, there is also a dependency regarding the number of entity instances that are part of the aggregate, since invocation of a microservice will bring an aggregate to memory [10]. This aspect is not addressed in this paper.
3. **Silhouette Score (SS)**, given by Eq. 4, where a represents the mean intra-cluster distance (Eq. 2: distance between object o_i and the remaining objects in the cluster) and b the mean nearest-cluster distance (Eq. 3: distance between object o_i and the objects of the neighbor cluster, the one that has the smallest average dissimilarity). This score ranges its values from -1 to 1, representing incorrect clustering (samples on wrong clusters) and highly dense clustering respectively. For every object in a cluster, when this score is high (closer to 1) the mean intra-cluster distance is going to be smaller than the mean nearest-cluster distance, implying that the object is well classified. This metric creates a parallelism with the overall coupling of the clusters of the system, as our objective was to obtain a high intra-cluster similarity and a low inter-cluster similarity, so the partition between clusters is well defined. The silhouette value evaluates exactly this information. In the scope of our problem we calculate the silhouette score for the entire cluster data of the presented cut, meaning that we have to calculate the silhouette of each cluster by averaging the score of all the object inside them and then average the score of all the clusters, reaching a value for the entire dataset.

$$a(o_i) = \frac{1}{|C_A| - 1} \sum_{o_j \epsilon C_A, o_j \neq o_i} d(o_i, o_j) \tag{2}$$

$$b(o_i) = min_{C_b \neq C_A} \frac{1}{|C_B|} \sum_{o_j \epsilon C_B} d(o_i, o_j) \tag{3}$$

$$Silhouette(o_i) = \frac{(b(o_i) - a(o_i))}{max(a(o_i), b(o_i))} \tag{4}$$

In Table 1 we apply the metrics for four cuts of a dendrogram with a max height of 4.0:

Table 1. Internal evaluation results for LdoD.

	Cut(0.01)	Cut(1.5)	Cut(2.5)	Cut(3.5)
Number of Retrieved Clusters (NRC)	40	11	3	2
Number of Singleton Clusters (NSC)	34	3	0	0
Maximum Cluster Size (MCS)	5	18	26	31
Silhouette Score (SS)	0.38	0.48	0.55	0.56

1. The maximization of intra-cluster similarity, given by a cut with the lowest possible value.
2. A cut at an intermediate value, establishing an attempt to make a trade-off between the granularity and the cluster similarity.
3. Two high valued cuts that try to split the system into its main components, usually with a size of 2–4 clusters.

Assessing first our ad-hoc metrics, when increasing the value of the height of the cut on the dendrogram, the NSC and NRC decrease while the MSC increases, which is expected as higher the height less clusters are formed, being that those contain more domain entities. Also, the silhouette score increases with height to a maximum, showing that at that point are formed the ideal clusters according to this metric.

External Evaluation. In this type of evaluation we compare with an expert decomposition both, the results of our approach and the results of applying a software architecture analysis tool, Structure101[6], which uses cyclomatic complexity measures and the identification of cyclic dependencies to define a structural decomposition.

Usually, the computation of evaluation metrics following the use of clustering is done in a pairwise fashion, where, in our case, the pairs of domain entities in the clusters of the decomposition being evaluated are compared with the pairs in the clusters of the domain expert decomposition. The most appropriate metrics for our approach are pairwise precision, recall and f-score, given by Eqs. 5, 6 and 7 respectively.

$$precision = \frac{tp}{tp + fp} \tag{5}$$

$$recall = \frac{tp}{tp + fn} \tag{6}$$

$$F\text{-}score = 2 * \frac{precision * recall}{precision + recall} \tag{7}$$

Where *tp* (true positives) represents the number of pairs that are in both, the decomposition being evaluated and the expert decomposition, *fp* (false positives) the number of pairs that are not in the expert decomposition but are in the decomposition being evaluated, and *fn* (false negatives) the number of pairs that are in the expert decomposition but not in the decomposition being evaluated. Therefore, the precision captures the accuracy whether two domain entities in a cluster actually belong to the same microservice, and the recall the percentage of domain entity pairs correctly assigned to the same microservice.

The pairwise assessment of these metrics for dendrogram cut of 2.5 is presented in Table 2, when comparing the two generated decompositions with the expert decomposition. We can see that the results of our approach for all the presented metrics are higher than Structure101. On the other hand, doing a detailed

[6] https://structure101.com/.

analysis, in the three clusters resulting from the 2.5 LdoD cut, we observe that the first is a sub-cluster of a cluster of the expert decomposition, the second is accessed by all controllers, and the third one contains five entities that are responsible for deleting and loading fragments, used by controllers associated with the administration functionalities. Structure101 originates ten clusters from which six are singletons. Of the remaining four, three are sub-clusters of the expert decomposition and the fourth is accessed by all controllers.

Table 2. External evaluation results for 2.5 cut, Structure101 and 1.5 cut.

	Precision	Recall	F-score
Transactional clustering 2.5 cut	73% ($\frac{445}{611}$)	48% ($\frac{445}{926}$)	0.58
Structure101	58% ($\frac{166}{285}$)	18% ($\frac{166}{926}$)	0.27
Transactional clustering 1.5 cut	99% ($\frac{233}{234}$)	25% ($\frac{233}{926}$)	0.40

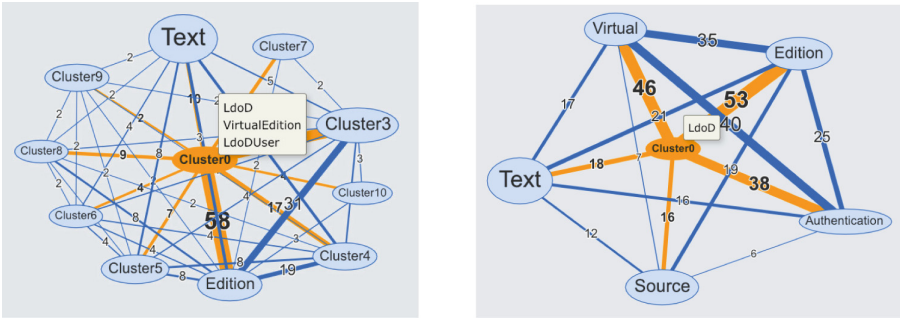
From this analysis we conclude that, although the use of metrics to identify the best cuts is relevant, it does not exclude the experimentation of other intermediate cuts because smaller clusters may be easily analysed by the expert, which may decided to integrate them with other clusters.

The chosen intermediate cut of the system and its evaluation is also shown in the Table 2. Note that for the 1.5 cut, our precision is much higher, this happens as the smaller clusters formed by a lower cut are almost all subsets of the clusters of the expert decomposition. On the other hand, the recall values are lower, as the singleton clusters are properly penalized by this metric. The only false positive (in 1/11 retrieved clusters) resides in the cluster with the entities `LdoD`, `LdoDUser` and `VirtualEdition`. `LdoD` is an immutable singleton and the entry point to the domain entities, which can be easily replicated in any cluster. `LdoDUser` and `VirtualEdition` were identified by the expert as being used in two different scopes, authentication and virtual edition management, respectively. Our tool classified these entities as being part of the same cluster as they appear together transactionally so, we are going to analyze these cases by using the visualization tool.

4.3 Visualization Analysis

After the metric evaluation of the clusters generated automatically, the software architect uses the visualization tool to do a detailed analysis of the decomposition.

Figure 2(a) shows Cluster0 containing the three entities. It has strong connections with other clusters, the edges thickness represent the number of controllers shared between the two connected clusters. Which means that almost all controllers access Cluster0. The model was already subjected to some changes by



(a) Initial Model (b) Manipulated Model

Fig. 2. Cluster views presenting clusters and the relations between them

the architect, basically, some of the clusters were renamed to have a domain-specific name, to improve readability. According to the expert these three entities, once created, are not further modified and are frequently accessed because they are the entry point for almost all functionalities. Therefore, since they are immutable, they can be easily replicated. This case constitutes a good example why the visualization tools provide an essential help to the software architect. Part (b) shows the model resulting from several transformation applied to the initial model, cluster rename, merge and split, and entity move between clusters, such that the architect can experiment, and fine tune, the decompositions.

Additionally, our visualization tool allows architects to identify how the business functionality is split between the different clusters. This is particularly relevant because it helps to analyze the impact of the decomposition in the business functionality.

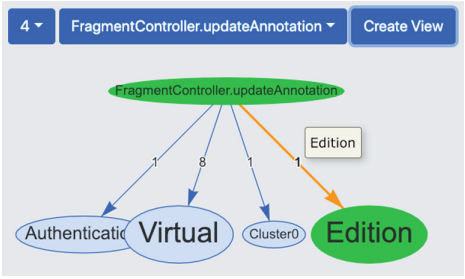


Fig. 3. Controller view of *updateAnnotation* controller and the clusters accessed.

Figure 3 shows the transactional behavior of `Update Annotation` controller occurring in the context of four clusters (candidate microservices). It is possible to identity which entities are accessed in each cluster, whose number is shown in

the edge. By analysing the model we can conclude that this decomposition does not have impact on the business logic of this functionality, because all semantically relevant accesses are to cluster Virtual. The accesses to the other clusters are to read immutable information for authentication (Authentication), access the persistent information through the `LdoD` singleton object (Cluster0), and get the `Edition` where the annotation is done (Edition). The figure highlights that the only entity accessed in cluster Edition is entity `Edition`. Note that this cluster contains more entities. It also illustrates that the controllers are selected by the number of clusters they access, 4 in this case (top left corner of the figure), which allows the software architect to easily identify in which controllers the decomposition can have more impact, if they access more clusters it may be necessary to relax their transactional behaviour.

Another visualization that can improve the split of functionalities is to identify which entities are accessed by controllers that also access other clusters, because it may be necessary to relax their consistency, since they are shared between business transactions executing through different microservices. However, when experimenting with this functionality, we realized that the each entity is accessed by all clusters, because there are some controllers, mainly administration controllers that create or delete the domain, and so, they access all domain entities. Therefore, we are considering, in future versions of the visualization tool, to allow the filtering of controllers, and also to use additional information to characterize the relations between clusters, for instance, by also collecting the dataflow between domain entities. Note that currently only the control flow information was collected.

From this discussion, we conclude that it is useful to analyse the relations between clusters through the use of our visualization tool, which shows that it is not enough to rely on the automatic decompositions, but tools should be provided to help to reason about the decomposition and its impacts, in particular, because the decomposition may have impact on the system business logic. Additionally, it is advantageous to enriched the visualization tool with modeling capabilities.

5 Related Work

In [12] it is done an analysis of several approaches for the migration of a monolith to a microservices architecture. Most microservices migration proposals do not consider the need to change the application business logic when migrating a monolith to a microservices architecture, focusing, instead, on the structural aspects related with the high cohesion and low coupling of the microservices. This problem is identified in [8] as the *Forgetting about the CAP Theorem* migration smell, and may have an high impact on the migration of a monolith because it imparts on the users perception of the system operation, which drives our decision to also provide tools for architectural experimentation.

In [23] the authors apply the three migration phases but the clustering phase is not automated, it is based on the observation of a graph. The data is collected

from use cases specifications and their decomposition into the domain entities they access. The metric is based on the data shared between operations, the operations that access the same data should belong to the same microservice. It is weighted by the reads and writes from the operations to the data objects, writes have more weight because there is a emphasis on having reads between microservices. They share with our approach the concern in focusing on how business functionalities are decomposed, but their final concern is on the operation level, instead of the controller, because they seek to have high cohesion and low coupling between operations. Their final visualization does not highlight how the business transactions are decomposed into the set of candidate microservices.

To improve performance, in [19], a runtime profiling is used to collect the information about the amount of communication between classes. Additionally, it also supports a semantic clustering that uses a *td-idf* (term frequency/inverse document frequency) to create clusters based on the similarity between names of classes and methods. None of these tactics consider transactional contexts and, so, the decomposition of the business logic. The user starts by deciding which of the two clustering criteria to use, and then visualizes the resulting graph, where a node represents a class and an edge a function call between two classes. Classes belonging to the same cluster have the same color and the edge thickness represents the amount of communication between classes. Representing clusters by colored classes has the advantage of making immediately visible the classes in a cluster, though it may the too confusing if there is a large number of classes. This is one of the few approaches that enhance visualization with modelling capabilities, it allows manipulation of the clusters, e.g. move a class between clusters, which results in the recalculation of the clusters, as we do.

In order to improve the performance of a microservices architecture, in [16], they apply a workload-based feature clustering. The approach is model-based, it uses a feature model, where the microservices identification, each microservice contains one or more features, is driven by a trade-off between performance, which is inversely proportional to the amount of inter-microservices communication, and scalability, which is directly proportional to the number of microservices. They propose the aggregation of features into microservices according to this trade-off in a specific-workload. They focus on feature model aggregation for deploy in a cloud instead of the identification of the microservices in a monolith, and consider that the implementation of the feature model allows features re-combinations, not considering how these impact on the application business logic, because different recombinations may impact differently.

In [9] it is proposed an approach for migrating a monolith implemented using Java Enterprise Edition (JEE). They do static analysis to capture the invocations between the elements. Afterwards, associate a cluster to each session bean and aggregate them according to a threshold, such that the distance between clusters depends on the number of shared entities. Final clusters have one or more session beans and the entities may be shared between different clusters. Finally, it is possible to visualize the clusters, showing the session beans it contains, and

the entities shared between two clusters. In our approach we aggregate the data entities that are accessed by the same business transactions, controllers, which is similar to their session beans, but their clusters are formed by session beans instead of domain entities, which hinders the analysis of how the business logic of a business transaction is split between microservices. Therefore, they assume that the microservices interfaces will be preserved, they correspond to the session beans interfaces, while we consider that the migration to the microservices architecture may impact on the application business logic due to the change in the overall consistency of the system, from strict to eventual or weak, which may require the carefully redefinition of the microservices interfaces.

In [1] each functional requirement is a microservice candidate. Afterwards they classify each candidate in terms of scalability and security non-functional requirements and the level of dependency between them. The candidates for which it is expected to have a high volume of requests are considered to require scalability. Then, for those with high and medium scalability requirements it is verified the level of dependency with the other candidates, where a high dependency level corresponds to the frequency of invocations between them. If two microservices are highly dependent and require security, which results in an high overheads, the candidates will be merged into a single microservice. This approach is model-based, which means that the data for the metrics is captured through requirements elicitation and focus on functional composition instead of on a real decomposition of a monolith.

In [15] execution traces analysis are used to generate two types of traces, class-level traces, which capture the classes accessed, and method-level traces, which capture the methods invocations. The microservices are identified by clustering the class-level traces that contain the same classes. Afterwards, the method-level traces are used to identify the interfaces for the candidate microservices. It does not propose any visualization tool. Similarly to our proposal, they aggregate the classes that are shared by the same business capabilities, contrarily to most approaches that focus on structural, coupling and cohesion, and semantic, naming convention, aspects. Their process is automatic, whereas we also propose a visualization tool that allows the experiment with several decompositions, to analyse the impact on the business logic.

In [2] they identify the microservices from a business process point of view. A business processes model is used to identify structural dependency, when there is a direct edge between two activities, and object dependency, when activities have similar data access, assigning a higher weight for writes. These two relations are aggregated in a metric that is use to generate clusters that represent candidate microservices. This model-based approach focus on the structural aspects, aggregate activities that access the same data and are executed next to each other, which result in clusters of activities from which is not possible to assess the impact of the decomposition on the application business logic. Actually, the business process model already describes a business logic between activities, and their aggregation may allow a more strict consistency between the activities

that become aggregated in the same microservice, as they will share the same transactional context. Their focus is on composition.

In [5] it is proposed a solution based on semantic similarity that matches the terms used in an OpenAPI specification with a domain vocabulary to suggest decompositions. This is a model-based approach, which requires two models (OpenAPI specification and domain vocabulary), and it is focused on identifying cohesive operations, which access the same data, ignoring the transactional business logic.

In [18] microservice candidates are suggested following an algorithmic approach subsequent to the extraction of data from a version control repository of a monolithic application. They propose three different metrics: single responsibility principle, based on classes that change together in commits; semantic coupling, based on *tfidf* to identify classes that contain code about the same things; and contributor coupling strategy, based on classes accessed by the same team. These metrics focus on the structural aspects, mainly related with the development process and the split of a domain model to control its complexity.

Some approaches propose the analysis of design trade-offs and the dynamic autonomous composition [14], but this is only applicable if the business logic does not vary according to the composition, which does not apply to all types of microservices. Therefore, the transactional contexts approach is particularly suitable for application with a rich domain logic where microservices become logically interdependent, which may require the redesign of the monolith functionality.

6 Conclusions

This paper proposes an approach to the migration of monolith applications to a microservices architecture that focus on the impact of the decomposition on the monolith business logic, an aspect that is not addressed by the existing approaches, and which is described as *forgetting about the CAP Theorem*. Our approach is based on the static analysis of the source code of a monolith, implemented following a Model-View-Controller architectural style, which is enforced by the most popular tools for web development, like Spring-Boot and Django. Therefore, a call graph is obtained for controllers, which are associated to the monolith functionalities. From the call graph are identified the domain entities that are accessed by each controller, and a clustering algorithm is applied to aggregate domain entities that are shared by the same controllers, to reduce the decomposition impact on the monolith business functionality. The resulting decomposition is analysed according to several metrics and an external evaluation, which compares the results with an existing industrial tool and a domain expert. The results are promising, but it is clear that it is necessary to provide more tools to support the experimentation with different candidate decompositions. Therefore, we also propose a visualization tool that allows the rename, merge and split of clusters, and the move of entities between clusters. It also supports different views, cluster, controller and entity, to help on the analysis of the impact of the decomposition on the monolith business logic.

Additionally, and due to the recent research done on the migration from monolith to microservices, the paper presents an extensive description of the related work in order to place our approach in a context that is quickly evolving and which is not yet completely bounded and classified.

The main limitations of this work are: (1) being specific for applications developed using the Fénix Framework; and (2) the java-callgraph tool did not capture calls inside Java 8 streams. Concerning the former limitation, we believe that the results apply to other implementations of web application, as soon as they clearly distinguish controllers from entities. Note that this also includes web applications that do not have views, but which provide a web API, e.g. REST. In what concerns the use of java-callgraph, we have done a manual verification of the collect data to ensure its correctness.

In terms of future work, we are already finishing an Eclipse plugin that captures the controllers call graphs using the Eclipse JDT library. On the other hand, we intend to experiment with decompositions where more information is available, in particular, we intend to distinguish reads from write accesses done by controllers and the dataflows inside controllers, to analyse its impact on cluster generations and in the visualization tools, because, in terms of eventual consistency of an application, the separation of reads from writes and dataflows are crucial for its software architecture design.

The tools source code is publicly available in a github repository[7].

Acknowledgment. This work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2019.

References

1. Ahmadvand, M., Ibrahim, A.: Requirements reconciliation for scalable and secure microservice (de)composition. In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), pp. 68–73, September 2016
2. Amiri, M.J.: Object-aware identification of microservices. In: 2018 IEEE International Conference on Services Computing (SCC), pp. 253–256, July 2018
3. Anquetil, N., Fourrier, C., Lethbridge, T.C.: Experiments with clustering as a software remodularization method. In: Proceedings of the Sixth Working Conference on Reverse Engineering, WCRE 1999, p. 235, IEEE Computer Society, Washington, DC (1999)
4. Ball, T.: The concept of dynamic analysis. SIGSOFT Softw. Eng. Notes **24**(6), 216–234 (1999)
5. 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
6. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 2000, p. 7. ACM, New York (2000)

[7] https://github.com/socialsoftware/mono2micro.

7. Cachopo, J., Rito-Silva, A.: Combining software transactional memory with a domain modeling language to simplify web application development. In: Proceedings of the 6th International Conference on Web Engineering, ICWE 2006, pp. 297–304. ACM, New York (2006)
8. Carrasco, A., van Bladel, B., Demeyer, S.: Migrating towards microservices: migration and architecture smells. In: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, pp. 1–6. ACM, New York (2018)
9. Escobar, D., et al.: Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American Computing Conference (CLEI), pp. 1–11, October 2016
10. Evans, E.J.: Domain-Driven Design: Tackling Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
11. Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA), p. 29-2909, April 2018
12. Fritzsch, J., Bogner, J., Zimmermann, A., Wagner, S.: From monolith to microservices: a classification of refactoring approaches. In: Bruel, J.-M., Mazzara, M., Meyer, B. (eds.) DEVOPS 2018. LNCS, vol. 11350, pp. 128–141. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-06019-0_10
13. 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
14. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: a self-adaptive roadmap. In: 2016 IEEE International Conference on Services Computing (SCC), pp. 813–818, June 2016
15. Jin, W., Liu, T., Zheng, Q., Cui, D., Cai, Y.: Functionality-oriented microservice extraction based on execution trace clustering. In: 2018 IEEE International Conference on Web Services (ICWS), pp. 211–218, July 2018
16. Klock, S., Van Der Werf, J.M.E.M., Guelen, J.P., Jansen, S.: Workload-based clustering of coherent feature sets in microservice architectures. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 11–20, April 2017
17. Maqbool, O., Babri, H.: Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. **33**(11), 759–780 (2007)
18. 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)
19. Nakazawa, R., Ueda, T., Enoki, M., Horii, H.: Visualization tool for designing microservices with the monolith-first approach. In: 2018 IEEE Working Conference on Software Visualization (VISSOFT), pp. 32–42, September 2018
20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6
21. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018)
22. Thönes, J.: Microservices. IEEE Softw. **32**(1), 116 (2015)
23. Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z.: Identifying microservices using functional decomposition. In: Feng, X., Müller-Olm, M., Yang, Z. (eds.) SETTA 2018. LNCS, vol. 10998, pp. 50–65. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99933-3_4