Automatic Microservices Identification from a set of Business Processes

Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, Aziz El Fazziki

Automatic Microservices Identification from a set of Business Processes

Mohamed Daoud[1], Asmae El Mezouari[2], Noura Faci[1], Djamal Benslimane[1]

Zakaria Maamar[3] and Aziz El Fazziki[2]

[1] Claude Bernard Lyon 1 University, Lyon, France
[2] Caddi Ayyad University, Marrakesh, Morocco
[3] Zayed University, Dubai, U.A.E

Abstract. All organizations engage in ongoing maintenance of their information systems due to constant changes in users' needs and governments' regulations. However these systems are monolithic making this maintenance a nightmare. To address this monolithic nature different technologies like commercial-of-the-shelf, service-oriented architecture, and lately microservices are proposed. This paper focuses on microservices by discussing their automatic identification from a set of business processes. Thanks to business processes, control and data dependencies between their activities are extracted and then clustered together. Each cluster constitutes a candidate microservice. To illustrate and demonstrate microservice automatic identification, a case study about renting bikes in the city of Barcelona is adopted and then implemented. In term of precision, the results show how business processes as inputs permit to generate better microservices compared to other approaches discussed in the paper, as well.

Keywords: Business Process · Control dependencies · Data dependencies · Microservice.

1 Introduction

Modern organizations' information systems (systems for short) are continuously subject to functional and architectural changes so these organizations can address users' new requirements and tap into the latest developments in Information and Communication Technologies (ICT). An example of functional change is when organizations deploy mobile services for users-on-the-move. And, an example of architectural change is when organizations adopt cloud for its elasticity and pay-as-you-go benefits. Although functional and architectural changes are a must, many organizations continue running their existing (legacy) systems despite the high maintenance cost along with missing the opportunities of improving their competitiveness posture. Many reasons justify this resisting "attitude" with focus in this paper on the **monolithic** nature of systems. Monolithic means one block that encompasses strongly-coupled components and that is sometimes "sealed" preventing its analysis for the sake of improvement.

Over the years many technologies were put forward to address monolithic systems including Commercial-Of-The-Shelf (COTS) [19], Component-Based Software Engineering (CBSE) [17], and lately Service-Oriented Architecture (SOA) [5]. Unfortunately many of these technologies did not live up to their expectations due to many reasons such as making organizations change to accommodate technologies, lack of capturing organizations' unique features, and complexity of adoption amplified with resistance to change. A recent trend referred to as microservices seems surging from the "ashes" of SOA [13].

Although microservices could be confused to SOA-based Web services and/or Restful services, Cerny et al. shed light on these 2 architectural styles' strengths and weaknesses in [3]. Both μServices with reference to Microservice Architecture and SOA have the same objective that is service cooperation through their integration across many independent platforms, but adopt different ways of achieving this cooperation. These architectures' strengths and weaknesses are related to 18 concerns that range from deployment and scalability to versioning and administration till business-rules location. While the 18 concerns provide a comprehensive coverage of how ICT practitioners could adopt either architecture, the identification of microservices and SOA services does not seem to be a concern. SOA is a well-established discipline so μServices could have benefited from its best practices when it comes to service identification, but this is not the case until now. In this paper we address this gap with a collaborative clustering-based approach to identify microservices from a set of Business Processes (BPs).

For a successful adoption of microservices, guidelines for identifying them are deemed necessary. The objective is to avoid system designers' disappointments in microservices, which could put them at risk of no-adoption like other technologies. The literature refers to different works on microservices identification using log files [8], source codes [9], UML class diagram [2], and legacy databases [4] as inputs to the identification exercise. While all these works have the same objective, automatic identification of microservices, none of them considers BPs as input. Referred to as organizations' main assets, BPs could constitute an important source for identifying microservices. According to Weske, *"... A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations"* [24]. Despite being a rich reservoir of many details like who does what, when, where, and why, BPs seem overlooked during the exercise of identifying microservices. To the best of our knowledge, Amiri is the only one who adopted BPs in this exercise [1].

Although the pioneering nature of Amiri's work, it suffers from many limitations. First, structural activity dependencies model is not defined to show how dependencies are formally obtained with respect to business process modeling languages' operators. Secondly, the data dependencies model is very simple and limited to read and write operations. Thirdly, only structural and data dependencies are considered while other dependencies like semantics between activities are overlooked. Fourthly, aggregating all dependency types into a single data structure (matrix in our case) to be used for clustering needs will lead to data quality degradation in the sense that strengths and/or weaknesses of each dependency type are potentially hidden.

In this paper, we design and develop a collaborative clustering-based approach to automatically identify microservices. At the core of our approach is the idea of extracting microservices from BPs' activities by using first, a set of separate models that individually identify different kinds of information related to these activities and their dependencies and second, a collaborative clustering technique instead of a classical clustering technique to avoid aggregating extracted data that could lead to losing some implicit details. Our approach is multi-models in the sense that it combines independent models to represent a BP's structural dependencies, data dependencies, semantic dependencies, and so on. It is also based on collaborative clustering in the sense that all data that could be extracted from the different models are kept independent instead of aggregating them. Each data set is then handled by a separate clustering algorithm. At the end, the collaborative clustering allows to each clustering component algorithm to benefit from the work done by other clustering components [6].

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 presents a case study, gives an overview of our approach to automatically identify microservices from existing BPs, and formalizes the structural and data dependencies models. Section 4 details the collaborative clustering in this approach and then, the system implementing this approach along with some experiments' results. Section 5 concludes the paper.

2 Related work

There exist a good number of works that discuss monolithic systems' limitations and how microservices could address them. These systems are known for incurring significant development, maintenance, and evolution cost [22].

In [11], a service-cutter system is presented allowing to decompose monolithic systems into small services. The decomposition uses the systems' functions and considers three criteria that are between entities at the property level, coherence between data of each microservice, and communication cost between services. In [4], an approach to identify microservices is proposed. The approach uses the semantic similarity of functionalities that are described in openAPI[4] and a reference vocabulary. Microservices are identified as a cohesive cluster of operations extracted from an UML diagram class. The semantic similarity is based on the pre-computed database DISCO (DIStributionally related words using CO-occurences)[5].

In [18], modernizing a monolithic system using microservices is proposed. This system is described using three types of objects: interfaces, business functions, and database tables. These objects are then linked in a dependency graph by means of calls from interfaces to business functions, calls between business functions themselves, and accesses from business functions to database tables. Candidate microservices correspond to the business rules that depend on database tables, and correspond to the facades connected to the database tables.

In [16], clustering techniques are used to optimize the performance and scalability of an existing microservice architecture. Given this architecture as an input, its current deployment, workload, features (a check of functionalities that deliver a business value) model that describes properties and their dependencies, an automatic approach is proposed to recommend a new deployment that optimizes the performance and scalability of the architecture. Some features can then be moved from one microservice to another. The results of experimenting the approach highlight the importance of considering performance metrics when generating a microservice architecture. A study that analyses microservices architectures with detailed performance profile data is also presented in [23].

In [14], a functionality-oriented microservice extraction by clustering execution traces of programs collected at run-time is described. These traces are collected by using techniques of program execution monitoring and are used to collect implicit and explicit program functional behavior. They also reveal which entities are used for which business logic. The approach clusters source code entities that are related to the same functionalities. Even if the work in [14] is interesting, it suffers from its strong dependence on the quality of the generated execution traces, and consequently on the quality of the test cases.

[4] www.openapis.org.
[5] www.commonspaces.eu/en/oer/disco-extracting-distributionally-related-words-us.

3 Our approach for identifying microservices

This section consists of 4 parts. The first part presents a case study that refers to Barcelona's bike sharing system known as Bicing. The second and third parts discuss our approach's foundations in terms of dependencies between BPs' activities and collaborative clustering. Finally, the last part formalizes these dependencies.

3.1 Bicing case-study

Bicing includes more than 400 bike anchor-stations spread across Barcelona and about 6000 bikes that users rent for a fee. Bicing's monolithic system is described in [10] along with the managerial and technical challenges that undermine its operations. For the needs of our work, we suggest in Fig. 1 a high-level representation of Bicing from a BP perspective. We resorted to the standard Business Process Model and Notation (BPMN) to illustrate this representation. We have identified different activities (a_1: request bike and a_9: dismantle bike), different dependencies (between a_1 and a_2), different logical operators (XOR between a_6 and a_{10} and OR between a_2 and a_3), and, finally, different artefacts (bike and user) and their respective attributes (e.g., ID and status). In the rest of this paper the discussion about Bicing is not restricted to renting bikes but includes other aspects like reporting and fixing bikes' defects and disposing bikes, if necessary. It all starts when a

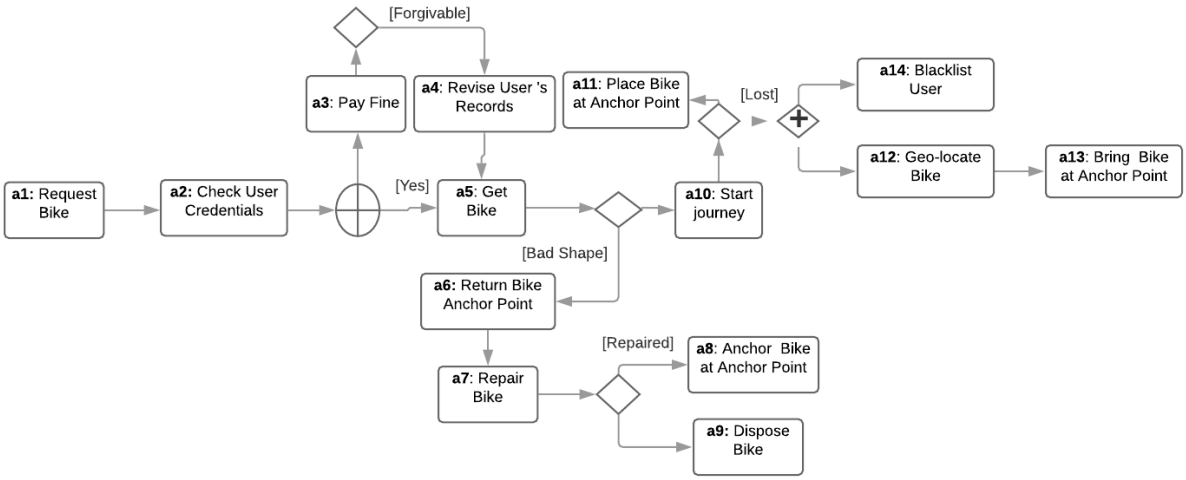


Fig. 1: An illustrative BPMN-based representation of the Bicing system

user requests a bike (a_1) at a certain anchor station. After checking the user's credentials (a_2) and any late fee payment (a_3), the Bicing system updates the user's records (a_4) and then, approves the user's request (a_5). If it turns out that the bike is defective, the user puts it back (a_6) and eventually requests another one. Otherwise, the user starts his journey (a_{10}). Regularly all bikes are serviced (a_7) leading to either putting them back for rent (a_8) or disposing them (a_9). When the user arrives to destination, he returns the bike at a certain anchor point (a_{11}). Otherwise, the Bicing

system blacklists the user (a_{14}) due to bike inappropriate return and geo-locates the bike (a_{12}) so that it is collected by the competent services and then made available to other users (a_{13}).

From a specification perspective, activities ($\{a_i\}$) may require inputs ($\{i_i\}$) and produce outputs ($\{o_i\}$) that both correspond to specific artefacts' attributes. An activity acts upon an attribute through 2 operations that are *read* (r) and/or *write* (w), which could lead to *updating* (u), *creating* (c), and/or *deleting* (d) artefacts. Table 1 lists activities, artefacts, attributes of artefacts, and also the operations that artefacts/attributes are subject to.

Table 1: Bicing's BPs as a set of activities, artefacts, attributes, and operations

Activity	Artefacts	Attributes
a_1	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Destination (r)
a_2	User (u)	User_ID (r), User_Credit (r), User_Destination (r)
a_3	User (u)	User_ID (r), User_History (r), User_Validity (r)
a_4	User (u)	User_ID (r), User_History (w)
a_5	Bike (u)	Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Status (w)
	Rental (c)	User_Destination (w), Rent_Date (w)
a_6	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Status (w)
	Rental (d)	Rent_ID (r)
a_7	Bike (u)	Bike_Status (w)
	Repair (c)	estimated_Repair_Cost (w), agree_Repair (w)
a_8	Bike (u)	Anchor_Point (r), Bike_Status (w)
a_9	Bike (d)	Bike_ID (r)
a_{10}	User (u)	User_ID (r), User_Status (w)
a_{11}	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r)
	Rental (u)	Rent_ID (r), Rent_Cost (w), User_History (w)
a_{12}	Bike (u)	Bike_ID (r), Bike_Location (w)
a_{13}	Bike (u)	Bike_ID (r), Anchor_Point (r), Bike_Status (w)
a_{14}	User (u)	User_ID (r), User_Status (w), User_History (w)

3.2 Foundations

It is largely known that BP automation helps organizations track events, assign activities, manage resources, etc. In this work we adopt the definition of BP given in [7] stating that a BP is a set of logically related activities that are performed to achieve goals. "Logically related" refers to dependencies between activities such as *control* (with respect to an execution order), data (with respect to information sharing), and *functional* (with respect to horizontal- and vertical-business

operations). By using BPs as an input to identifying microservices, we would like to ensure that these microservices are fine-grained, strongly cohesive (i.e., degree to which activities in a microservice belong together), and loosely-coupled (i.e., degree to which microservices can be easily replaced). These 3 dependencies are illustrated below.

- *Control dependency* refers to both the execution order (e.g., finish-to-start and start-to-start) between activities and the logical operators (e.g., XOR and AND) between activities as well. Should 2 activities be directly connected through a control dependency, then most probably they would form a highly-cohesive microservice to which they will belong. Contrarily, they would most probably be used to form separate microservices to which each will belong.
- *Data dependency* refers to associating activities' outputs/inputs in a way that permits to illustrate data flowing from one activity to another. These inputs/outputs correspond to artefacts' attributes. Data dependency sheds light on both artefacts and artefacts' attributes that could be subject to operations illustrated in Table 1 like *create* (c) and *write* (w). In addition to input/output association, data dependency could indicate to what extent artefacts and/or artefacts' attributes are either mandatory or optional for BP execution. We advocate that activities that exchange mandatory artefacts' attributes should be part of the same microservice allowing to avoid delaying this exchange, for example.
- *Functional dependency* refers to horizontal (cross) and vertical (silo) business operations. The former denotes activities whose execution would cross multiple departments in the same organization. The latter denotes activities whose execution would be confined into the same department. We advocate that activities that would take part in horizontal interactions would less likely be part of the same microservices.

We focus on *control* and *data dependencies*, only. Upon establishing such dependencies, we quantify them using specific metrics. The objective is to evaluate cohesion and coupling among activities so that they are gathered in either same or separate microservices. To measure a *control dependency* between 2 activities (a_i, a_j), we consider a_j's occurrence probability after executing a_i. This probability depends on the execution order and/or logical operators between a_i and a_j. Let us consider the *control dependency* between a_5 and a_{10} that is connected with a_6 through XOR (Fig. 1). After executing a_5, a_{10}'s occurrence probability depends on the decision made at XOR (i.e., either a_6 or a_{10}). We note that any activity's occurrence probability is calculated over time by using BP's execution logs. To measure a *data dependency* between 2 activities (a_i, a_j), we consider an artefact's and attribute's criticality level that would reflect the importance of information shared between these activities. This level denotes the impact of artefact/attribute unavailability on the continuity of business operations. More details are given in the next sections.

Based on dependencies among activities, we gather activities into microservices by using clustering techniques. In the literature, clustering is either centralized or collaborative [12]. In the former, a single component manages the clustering by utilizing all individuals' features[6] as inputs. In the latter, multiple components, each in charge of one type of features, exchange some details during clustering so that appropriate clusters are jointly built. Performance and appropriateness of clustering techniques are thoroughly discussed in the literature [6] and [25]. Many works like [6] and [12] advocate for collaborative clustering to identify microservices. It provides fine-grained and accurate results contrarily to centralized clustering where individuals' features need to be aggregated before initiating any clustering.

[6] In our work, individuals are activities and features are control and data dependencies.

Fig 2 depicts our approach for microservices identification. It relies on the aforementioned dependencies and collaborative clustering to group activities that would form fine-grained, highly-cohesive, and loosely-coupled microservices. Our approach consists of the following steps. After examining activity dependencies, their details are stored in dedicated repositories. Then, these repositories' contents are submitted for collaborative clustering where different clustering techniques (one per dependency) are used to obtain consensual clustering solutions. More details are given in Section 4.1.

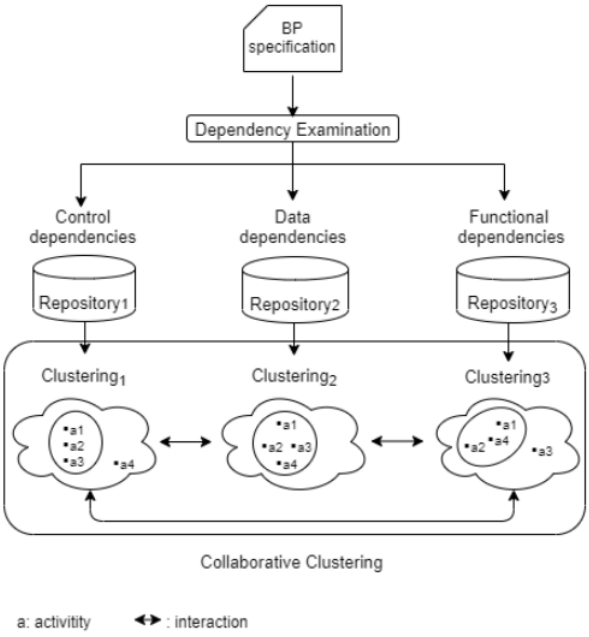


Fig. 2: General representation of our microservice identification approach

3.3 Control dependency analysis

Let CD(a_i, a_j[Operator $\{a_k\}$])$_{\text{ExecOrder}}$ be a a direct *control dependency* referring to a certain execution order between a_i and a_j that is connected to other activities $\{a_k\}$ through a certain Operator. An execution order between 2 activities could be exemplified with either *finish-to-start* (our focus and denoted by SEQ)), *finish-to-finish*, *start-to-start*, or *start-to-finish*.

Let us start with the *control dependency* $CD(a_i, a_j)_{\text{SEQ}}$ (i.e., $\{a_k\} = \emptyset$)). Since SEQ between a_i and a_j means that a_j starts only after a_i has successfully completed, $CD(a_i, a_j)_{\text{SEQ}}$ denotes a_j's occurrence probability (p) after a_i's completion as per Equation 1:

$$CD(a_i, a_j)_{\text{SEQ}} = p \tag{1}$$

where $p \in]0, 1]$.

We now examine the *control dependency* $CD(a_i, a_j \ \mathsf{Operator}\ \{a_k\})_{\mathsf{SEQ}}$ (i.e., $\{a_k\} \neq \emptyset$)). According to Operator's semantics, we assume that some r activities in $\{a_k\} \cup a_j$ will be selected for execution. Equation 2 defines the number of activities that will be selected for execution as a combination $C(n, r)$ where n corresponds to $card(\{a_k\} \cup a_j)$.

$$C(n, r) = \frac{n\,!}{r\,! \times (n - r)\,!} \tag{2}$$

Depending on the semantics of Operator whether AND, XOR, or OR, $CD(a_i, a_j \ \mathsf{Operator}\ \{a_k\})_{\mathsf{SEQ}}$ is calculated as follows:

1. $CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that a_j will start only after a_i has successfully completed regardless of $\{a_k\}$. Formally, Equation 3 computes $CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}}$ as follows:

$$CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}} = C(n, n) * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{3}$$

   where $p \in]0, 1]$ & $C(n, n) = 1$, as per Equation 2.
2. $CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that one activity from $\{a_k\} \cup a_j$ will be selected after a_i has successfully completed. Formally, Equation 4 computes $CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}}$ as follows:

$$CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}} = \frac{1}{C(n, 1)} * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{4}$$

   where $C(n, 1)$ is the number of possibilities to select one activity from $\{a_k\} \cup a_j$. As per Equation 2, $C(n, 1)$ is equal to n.
3. $CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that a set of r activities from $2^{\{a_k\} \cup a_j}$ (i.e., all possible multiple choices) will be selected after a_i has successfully completed. For the sake of simplicity, we assume that any activity in $\{a_k\} \cup a_j$ has the same occurrence probability over $2^{\{a_k\} \cup a_j}$, that is equal to $\frac{r}{n}$ where r varies from 1 to n. Formally, Equation 5 computes $CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}}$ as follows.

$$CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}} = \frac{\sum_{r=1,n}(\frac{r}{n} \times C(n, r))}{\sum_{r=1,n} C(n, r)} * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{5}$$

   where
   - $\sum_{r=1,n}(\frac{r}{n} \times C(n, r))$ represents the number of a_j's occurrences among possible combinations of activities[7].
   - $\sum_{r=1,n} C(n, r)$ corresponds to the total number of possible combinations of activities[8].

We now look into indirect *control dependency* between a_i and a_j where there is a set of other peers connected with operators. This dependency, denoted as $CD(a_i, a_j)_{path^1_{i,j}}$, refers to a certain execution path ($path^1_{i,j}$) and is computed as per Equation 6:

$$CD(a_i, a_j)_{path^1_{i,j}} = \prod_{a_l, a_m \in path^1_{i,j}} CD(a_l, a_m \ \mathsf{Operator}\ \{a_{k_m}\})_{\mathsf{SEQ}} \tag{6}$$

[7] Let n be 3, $\sum_{r=1,n}(\frac{r}{n} \times C(n, r))$ has the following value: $(\frac{1}{3} \times 3 + \frac{2}{3} \times 3 + \frac{3}{3} \times 1{=}4)$.
[8] Let n be 3, $\sum_{r=1,n} C(n, r)$ has the following value: (3+ 3+ 1)=7.

When multiple execution paths exist between a_i and a_j, we refer to this *control dependency* as $CD(a_i, a_j)_{paths_{i,j}}$ and is computed as per Equation 7:

$$CD(a_i, a_j)_{paths_{i,j}} = \max_{q=1,...} (CD(a_i, a_j)_{path^q_{i,j}}) \tag{7}$$

where $path^q_{i,j}$ represents the q^{th} possible execution path between a_i and a_j. Table 2 depicts an excerpt of *control dependencies* in the Bicing system.

Table 2: Control dependencies with $p = 0.5$

Activity	a_1	a_2	a_3	a_4	a_5
a_1	-	1/2	5/6	11/12	17/12
a_2	1/2	-	1/3	7/12	11 /12
a_3	5/6	1/3	-	1/4	3/4
a_4	11/12	7/12	1/4	-	1/2
a_5	7/12	11/12	3/4	1/2	-

3.4 Data dependency analysis

As mentioned in Section 3.2, we consider artefact's and attribute's *criticality* as a metric for measuring *data dependency* between 2 activities. In [21], Paulsen et al. describe a comprehensive criticality analysis for helping organizations identify and prioritize assets (e.g., artefacts and processes) that are vital for the success of organizational goals. Based on this analysis, we distinguish 2 types of criticality: *functional* and *non-functional*. The former refers to artefact/attribute's unavailability that would hinder the BP's proper execution while the latter refers to artefact/attribute's corruption would undermine the BP's QoS. On the one hand, we rely on the traditional 3 information levels (i.e., *strategic*, *tactical*, and *operational*) to define artefact's and attribute's *functional-criticality* level. A *strategic* attribute's criticality should be higher than an *operational* attribute. On the other hand, we rely on security, privacy, and safety levels to define artefact's and attribute's *non-functional-criticality* level. Table 3 depicts certain attributes per type of criticality for the Bicing case-study.

Table 3: Attribute examples per criticality type

Type	Level	Example
Functional	Operational	Decision-making data
	Tactical	Concurrency data
	Strategic	Customer experience data
Non-functional	Privacy	Protected personal data
		Stakeholder's identity
	Confidentiality	Financial data

We define 3 criticality degrees, *high* (H), *medium* (M), and *low* (L) that have, respectively, k, $k' < k$, and $k'' < k'$ as quantitative value. Note that the BP designer specifies artefact's and attribute's criticality level with respect to Table 3. Since artefacts and attributes can be associated with both types of criticality (i.e., *functional* and *non-functional*), we define 2 strategies for calculating an artefact (ar)'s and attribute (at)'s criticality ($\mathcal{C}(ar|at)$). The first strategy computes $\mathcal{C}(ar|at)$ as a weighted sum (Equation 10).

$$\mathcal{C}(ar|at) = w_1 \times \mathcal{C}^F(ar|at) + w_2 \times \mathcal{C}^{NF}(ar|at), \quad w_1 + w_2 = 1 \tag{8}$$

where:

- $\mathcal{C}^F(ar|at) \mid \mathcal{C}^{NF}(ar|at)$ corresponds to $ar|at$'s *functional/non-functional criticality* degree, and
- $w_1 \mid w_2$ is the weight (i.e., importance) the BP designer associates with $\mathcal{C}^F(ar|at) \mid \mathcal{C}^{NF}(ar|at)$.

Considering k, k', and k'' as parameters gives more flexibility and generality to our approach. Some simulations will show how k's and k''s values impact the cohesion among activities within the identified microservices and fine-coupling among those microservices.

Table 4: Artefact/Attribute criticality for Bicing case-study

Artefact	Attributes	$\mathcal{C}^F$
Bicycle	Anchor_Point	M (k'$_1$)
	Bike_Status	H (k$_1$)
User	User_Status	H (k$_2$)
	User_Destination	L (k"$_2$)
	User_History	H (k$_2$)
Rental	Rent_ID	H (k$_3$)
	Rent_Cost	M (k'$_3$)
Repair	agree_Repair	M (k'$_4$)

Artefact	Attributes	$\mathcal{C}^{NF}$
Bicycle	Bike_ID	H (k$_1$)
User	User_ID	H (k$_2$)
	User_Validity	M (k'$_2$)
Repair	estimated_Repair_Cost	H (k$_4$)

Once $ar|at$'s criticality is established, we specify *data dependencies* ($\mathcal{DD}$) between a_i and a_j as follows:

$$\mathcal{DD}(a_i, a_j) = \sum_{d_p \in DATA_{i,j}} pair(d_p)_{i,j} \times \mathcal{C}(d_p) \tag{9}$$

where

- $DATA_{i,j}$ indicates the set of artefacts/attributes exchanged between a_i and a_j,
- d_p represents the artefact/attribute exchanged between a_i and a_j,
- $pair(d_p)_{i,j}$ denotes the value associated with the operation pair (e.g., r/w and w/w) between a_i and a_j, proposed by Amiri [1].

The second strategy considers $\mathcal{C}(ar|at)$ as a tuple $< \mathcal{C}^F(ar|at), \mathcal{C}^{NF}(ar|at) >$. We, thus, specify *data dependencies* ($\mathcal{DD}$) between a_i and a_j as follows:

$$\mathcal{DD}(a_i, a_j) = \sum_{d_p \in DATA_{i,j}} \mathcal{F}(pair(d_p)_{i,j}, \mathcal{C}^F(d_p), \mathcal{C}^{NF}(d_p)) \tag{10}$$

where $\mathcal{F}$ returns the *data dependency* value specified by the BP designer for the tuple $< pair(d_p)_{i,j}, \mathcal{C}^F(d_p), \mathcal{C}^{NF}(d_p) >$.

Table 5: Excerpt of data dependencies for the 1^{st} strategy

Activity	a_1	a_2	a_3	a_4	a_5
a_1	-	1/2	5/6	11/12	17/12
a_2	1/2	-	1/3	7/12	11 /12
a_3	5/6	1/3	-	1/4	3/4
a_4	11/12	7/12	1/4	-	1/2
a_5	7/12	11/12	3/4	1/2	-

4 Experimenting the collaborative clustering

In this section we detail the collaborative clustering approach and then present how it was implemented and evaluated.

4.1 Collaborative clustering

Clustering is about partitioning a set of objects into groups called clusters. Each cluster regroups similar objects in the sense that objects of a group are more similar to each other than objects of other groups. In our approach, objects represent activities "expecting" that activities of the same group would form a highly-cohesive candidate mircoservice. Contrarily, activities in different groups would form loosely-coupled microservices.

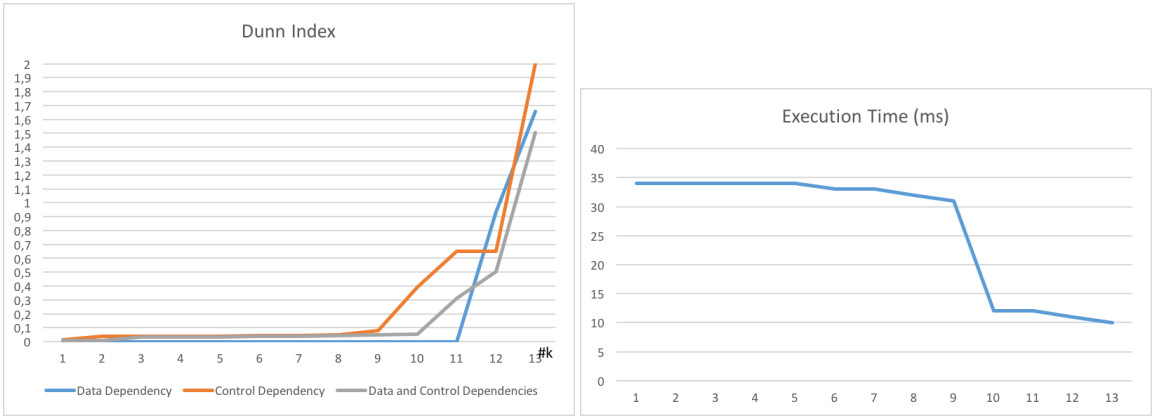
We extended the classical Hierarchical agglomerative algorithm (HAC) [20] to design our collaborative clustering algorithm (cHAC). cHAC runs over n clustering nodes ($CN_1, CN_2, , CN_n$), where a CN partitions the whole set of activities into clusters of activities by using one given dependency matrix. The chosen number k of clusters at each CN is not necessary the same; it can be different from one CN to another. cHAC fosters collaboration between CN since each CN will have its own dependency matrix along with "keeping an eye" on what other CNs are doing by sharing some dependencies scores of activities, if deemed necessary. Thus, prior to each new HAC clustering iteration, a CN uses both a Local Score Matrix (LSM) that stores dependency scores between couple of activities and a Shared Score Matrix (SSM) that stores a global dependency score between each couple of activities.

- cHAC's first iteration: creation of the local dependence score matrix. It corresponds to the dependency matrix used as input. An empty shared dependence matrix is also created to store the shared dependency score of activities. Each activity constitutes a cluster.
- cHAC's p^{th} iteration: the nearest pair of clusters C_u and C_v is computed by using both LDSM and SDSM. C_u and C_v are merged if and only if[9] $[distance(C_u, C_v)]^p_{LDSM} >= [distance(C_u, C_v)]^{p-1}_{SDSM}$. To foster similarities between couples of activities (t_i, t_j), the shared score matrix is also updated as follows: $[LDSM(t_i, t_j)]_p = Max([LDSM(t_i, t_j)]_p, [MSP(t_i, t_j)]_{p-1})$

[9] $distance[distance(C_u, C_v)]^p_M$ computes any distance between 2 clusters C_u and C_v by using the score Matrix M of the iteration p.

Once the different clustering results are produced by the different CNs, the distance metrics are applied to them to choose the best one that fosters both cohesion and loose-coupling of groups. It is important to note that our cHAC algorithm can work either in a uniform collaboration strategy whereh each CN collaborates with other CNss, or in diverse collaboration strategy where each CN node has its own collaborators. For the latter case, different shred matrices are needed, one by CN node.

Our cHAC is different from the distributed HAC (dHAC) [15]. dHAC consists of 2 phases. In the first phase, the entire collection of objects is divided into n disjoint segments and distributed over n HAC processes. Each HAC process generates a separate clustering. In the second phase, the previous generated clusters are merged into one final cluster result. In our cHAC, and contrarily to dHAC, the collaboration between the different HAC nodes is realized between two successive iterations by sharing intermediate clustering results.



(a) Dunn Index of the clustering results (b) Execution time of the clustering algorithm

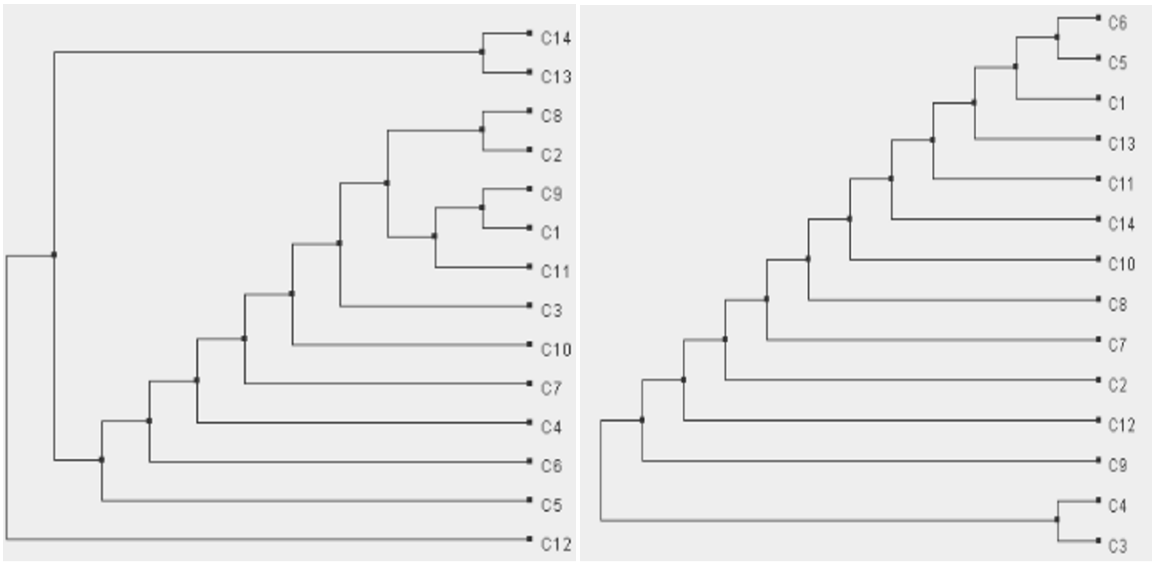
Fig. 3: Bicing system's 14 activities analysis

4.2 Experiments

We implemented the collaborative clustering approach in Java. To this end, different modules have been developed. Some of these modules permitted to extract control dependencies and data dependencies from BPs and to run the collaborative clustering algorithm that took the number of clusters and dependency matrices as inputs. Bicing system's 14 activities were initially used to test the algorithm. Then, more activities were randomly generated to capture the complexity of real BPs.

For evaluation needs we considered the internal validation metric Dunn index that measures the quality of clustered results by identifying the clusters that are compact (minor variance between activities of the same cluster) and separate (clusters are enough far apart). A higher Dunn index indicates better clustering.

In the first experiment, we measured the Dunn index of the clustering algorithm with respect to the "Control Dependencies" CN node (CDCN) and the "Data Dependencies" CN node (DDCN).



(a) Dendogram at the control dependency CN node (b) Dendogram at the data dependency CN node

Fig. 4: Dendograms at CN nodes

Fig. 3a illustrates the obtained results clearly showing that the Dunn index that results from the CDCN node is almost always better than the Dunn index that results from the DDCN node. This means that for a given BP, the control dependency model is richer and more informative than the data dependency model. We also computed the Dunn index of the clustering result by aggregating both control and data dependencies in one matrix.

Fig. 3b illustrates the obtained results showing that the clustering quality of the CDCN node is often better than the one obtained by merging control and data dependencies. It confirms that aggregating different dependency matrices degrades the quality of the final microservice generation and the collaboration between nodes is the appropriate option. Fig. 4 illustrates the obtained dendograms at both control and data dependency nodes. C_i refers to teh activity a_i.

5 Conclusion

To address monolithic systems' limitations, this paper examined microservices as a novel way for breaking down these systems into "small" manageable units. We shed light on how to identify necessary microservices from a set of BPs. Compared to other approaches that adopt log files, source codes, among others as inputs to identifying microservices, BPs constitute a better alternative to these inputs. Indeed, referred to as organizations' know how, BPs' activities allow to define who does what, when, where, and why. We firstly capture these details into different kinds of separate dependencies known as control and data. We then process these dependencies using our designed collaborative clustering algorithm. Each cluster of activities constitutes a microservice candidate.

The technical doability of our work has been verified using the Bicing system that allows users to rent bikes in the city of Barcelona. The results are promising showing how our collaborative

clustering algorithm outperforms in term of precision the cooperative clustering algorithm. In term of future work we would like to consider a larger dataset for testing the collaborative clustering algorithm, examine other forms of dependencies like semantics dependencies between activities, benchmark our identified microservices to other approaches that do not use BPs, and finally recommend clustering techniques with respect to the nature of these dependencies.

References

1. Amiri, M.J.: Object-aware identification of microservices. In: 2018 IEEE International Conference on Services Computing (SCC). pp. 253–256. IEEE (2018)
2. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: European Conference on Service-Oriented and Cloud Computing. pp. 19–33. Springer (2017)
3. Cerný, T., Donahoo, M.J., Pechanec, J.: Disambiguation and comparison of soa, microservices and self-contained systems. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS 2017, Krakow, Poland, September 20-23, 2017. pp. 228–235 (2017)
4. Chen, R., Li, S., Li, Z.: From monolith to microservices: A dataflow-driven approach. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). pp. 466–475. IEEE (Dec 2017)
5. Chung, J., Chao, K.: A view on service-oriented architecture. Service Oriented Computing and Applications **1**(2), 93–95 (2007). https://doi.org/10.1007/s11761-007-0011-2, `https://doi.org/10.1007/s11761-007-0011-2`
6. Cornuéjols, A., Wemmert, C., Gançarski, P., Bennani, Y.: Collaborative clustering: Why, when, what and how. Information Fusion **39**, 81–95 (2018). https://doi.org/10.1016/j.inffus.2017.04.008, `https://doi.org/10.1016/j.inffus.2017.04.008`
7. Davenport, T., Short, J.: The new industrial engineering: Information technology and business process redesign. Sloan Management Review (1990)
8. Djogic, E., Ribic, S., Donko, D.: Monolithic to microservices redesign of event driven integration platform. In: 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018. pp. 1411–1414 (2018). https://doi.org/10.23919/MIPRO.2018.8400254, `https://doi.org/10.23919/MIPRO.2018.8400254`
9. Escobar, D., Cardenas, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., Casallas, R.: Towards the understanding and evolution of monolithic applications as microservices. In: XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile, October 10-14, 2016. pp. 1–11 (2016). https://doi.org/10.1109/CLEI.2016.7833410, `https://doi.org/10.1109/CLEI.2016.7833410`
10. Estañol, M.: Artefact-centric Business Process Models in UML: Specification and Reasoning. Ph.D. thesis, Universitat Politècnica de Catalunya (2016)
11. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: A systematic approach to service decomposition. In: European Conference on Service-Oriented and Cloud Computing. pp. 185–200. Springer (2016)
12. Hammouda, K., Kamel, M.: Collaborative document clustering. In: SIAM International Conference on Data Mining (2006)
13. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: A self-adaptive roadmap. In: IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016. pp. 813–818 (2016). https://doi.org/10.1109/SCC.2016.113, `https://doi.org/10.1109/SCC.2016.113`
14. 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 2018, San Francisco, CA, USA, July 2-7, 2018. pp. 211–218 (2018). https://doi.org/10.1109/ICWS.2018.00034, `https://doi.org/10.1109/ICWS.2018.00034`

15. Ke, W., Gong, X.: Collaborative hierarchical clustering in the browser for scatter/gather on the web. In: Information, Interaction, Innovation: Celebrating the Past, Constructing the Present and Creating the Future - Proceedings of the 75th ASIS&T Annual Meeting, ASIST 2012, Baltimore, MD, USA, October 26-30, 2012. pp. 1–8 (2012). https://doi.org/10.1002/meet.14504901139, https://doi.org/10.1002/meet.14504901139
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 2017, Gothenburg, Sweden, April 3-7, 2017. pp. 11–20 (2017). https://doi.org/10.1109/ICSA.2017.38, https://doi.org/10.1109/ICSA.2017.38
17. Kouroshfar, E., Shahir, H.Y., Ramsin, R.: Process patterns for component-based software development. In: Component-Based Software Engineering, 12th International Symposium, CBSE 2009, East Stroudsburg, PA, USA, June 24-26, 2009, Proceedings. pp. 54–68 (2009). https://doi.org/10.1007/978-3-642-02414-6_4, https://doi.org/10.1007/978-3-642-02414-6_4
18. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint arXiv:1605.03175 (2016)
19. Mendez-Bonilla, O., Franch, X., Quer, C.: Requirements patterns for COTS systems. In: Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), February, 25-29, 2008, Madrid, Spain, Proceedings. pp. 232–234 (2008). https://doi.org/10.1109/ICCBSS.2008.34, https://doi.org/10.1109/ICCBSS.2008.34
20. Murtagh, F., Legendre, P.: Ward's hierarchical clustering method: Clustering criterion and agglomerative algorithm. CoRR **abs/1111.6285** (2011), http://arxiv.org/abs/1111.6285
21. Paulsen, C., J.M., B., Bartol, Winkler, N.: Criticality analysis process model. Tech. rep. (2018)
22. Singh, V., Peddoju, S.K.: Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation (ICCCA). pp. 847–852. IEEE (2017)
23. Ueda, T., Nakaike, T., Ohara, M.: Workload characterization for microservices. In: 2016 IEEE international symposium on workload characterization (IISWC). pp. 1–10. IEEE (2016)
24. Weske, M.: Business Process Management - Concepts, Languages, Architectures,-driven. 2nd edition, Springer (2012)
25. Wilkin, G., H., X.: A practical comparison of two k-means clustering algorithms. BMC bioinformatics **9** (2008)