Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems

Adambarage Anuruddha Chathuranga De Alwis[1], Alistair Barros[1(✉)], Artem Polyvyanyy[2], and Colin Fidge[1]

[1] Queensland University of Technology, Brisbane, Australia
{adambarage.dealwis,alistair.barros,c.fidge}@qut.edu.au
[2] The University of Melbourne, Parkville, VIC 3010, Australia
artem.polyvyanyy@unimelb.edu.au

Abstract. We present heuristics that help to identify suitable consumer-oriented parts of enterprise systems which could be re-engineered as microservices. Our approach assesses the key structural and behavioural properties common to both enterprise and microservice systems, as needed to guide a microservices discovery process and coherently assess restructuring recommendations. Building upon existing business object and system structural definitions, we present heuristics for two fundamental areas of microservice discovery, namely function splitting based on object subtypes (i.e., the lowest granularity of software based on structural properties) and functional splitting based on common execution fragments across software (i.e., the lowest granularity of software based on behavioural properties). A prototype analysis tool was developed based on the defined heuristics and experiments show that it can identify microservice designs which support multiple microservice characteristics, such as high cohesion, low coupling, high scalability, high availability, and processing efficiency while preserving coherent features of enterprise systems. In particular, we illustrate the usefulness of this new approach by conducting a case study based on customer management systems: SugarCRM and ChurchCRM.

Keywords: Microservice discovery · System reengineering
Cloud migration

1 Introduction

Microservices have emerged as the latest style of service-based software allowing systems to be distributed through the cloud as fine-grained components, typically with individual operations, in contrast to services under a Service-Oriented Architecture (SOA) which include all logically related operations [1]. As such, microservices allow specific parts of systems and the business processes they support, down to individual tasks, to be scaled up and replicated through the

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 37–53, 2018.
https://doi.org/10.1007/978-3-030-03596-9_3

cloud, and be flexibly composed in Web, mobile computing, and Internet-of-Things (IoT) applications. These benefits originally led Netflix, and now Twitter, eBay, Amazon and other Internet players, to develop novel architectures for software solutions as microservices. Nonetheless, microservices have so far not been adopted for the dominant form of software in businesses, namely enterprise systems, limiting such systems' evolution and their exploitation of the full benefits of cloud-enabled platforms such as Google Cloud, Amazon AWS and IoT [2].

Enterprise systems, such as enterprise resource planning (ERP), customer relationship management (CRM) and supply chain management are large and complex, and contain complex business processes encoded in application logic managing business objects (BOs), in typically many-to-many relationships [3]. Restructuring enterprise systems as microservices is technically cumbersome, requiring tedious search and identification of suitable parts of the system to restructure, program code rewrites, and integration of the newly developed microservices with the 'backend' enterprise systems. This is a costly and error-prone task for developers, because enterprise systems have millions of lines of code and thousands of BOs they manage, entailing a multitude of functional dependencies, in and across many software packages and modules. In addition, microservices are the most fine-grained and loosely-coupled form of software components upon which to restructure large-scale enterprise systems. This leads to major uncertainties about the best way to split enterprise systems functions as microservices, to achieve high scalability and availability and low system latencies through the cloud, while attaining high cohesion and low coupling between software components.

Automated software re-engineering techniques have been proposed to improve the efficiency of transforming legacy applications, addressing specifically cohesion and coupling of software packages and components using static analysis techniques that focus on source code and dynamic analysis techniques that focus on software execution recorded in system logs. Even though these analyses proposed to improve software search and metrics, studies show that the success rate of software re-modularisation techniques, especially for large systems, remains low [4]. The key stumbling blocks are the limited insights available from syntactic structures of software code for profiling software dependencies and not identifying the semantics available through the business object relationships [5].

Enterprise systems can provide enriched semantic insights, available through the BOs that they manage which influence the software structure and the processes they support. For instance, an order-to-cash process in SAP ERP is supported through functions of software components: multiple sales orders, deliveries shared across different customers, shared containers in transportation carriers, and multiple invoices and payments. To support this process, multiple functions are invoked asynchronously, reflecting BO relationship types and cardinalities, and are seen through cross-service interactions, correlations, and data payloads [6]. Such insights provided by BO relationships are promising for improving the feasibility of automated discovery applications. As examples, Pérez-Castillo *et*

al. [7] used transitive closures of strong BO dependencies derived from databases to recommend software functions hierarchies, while Lu *et al.* [8] demonstrated process discovery using SAP ERP logs based on BOs.

This paper presents discovery techniques that help to identify suitable consumer-oriented parts of enterprise systems which could be re-engineered as microservices with desired characteristics such as high cohesion, low coupling, high scalability, high availability and high processing efficiency. It does so by providing an abstraction of the systems architecture, using key structural and behavioural properties common to both enterprise and microservices systems, considered essential to guide microservices discovery processes and coherently assess their potential restructuring. The structural properties address the functional composition of software, namely functions and their BO Create-Read-Update-Delete (CRUD) operations, while behavioural properties focus on system executions, at the level of operation invocations, reflecting single-entry-single-exit sequences characteristic of these systems. This, in principle, allows enterprise systems to be analysed at different units of structural and behavioural granularity, and the resulting restructure recommendations to be conveniently assessed for preservation of structural and behavioural properties. This paper addresses two fundamental areas of microservice discovery, namely function-splitting based on object subtypes (i.e., the lowest granularity of software based on structural properties) and functional splitting based on common execution fragments across software (i.e., the lowest granularity of software based on behavioural properties). This, we argue, provides a solid basis for future development of further microservices discovery heuristics.

The remainder of the paper is structured as follows. Section 2 presents structural and behavioural properties of software systems, while Section 3 exploits these properties to propose heuristics for discovering microservises in enterprise systems. Section 4 discusses an implementation and validation of the proposed heuristics. Related work is summarized in Section 5. The paper closes with a conclusion.

2 Structural and Behavioural Properties of Enterprise and Microservice Systems

This section describes the essential properties of a target system architecture that comprise an enterprise system (ES) and a microservice (MS) system, which is depicted in Fig. 1. This architecture will be used in our MS discovery approach (detailed in Sect. 3). The architecture reflects a unified software structure for both an ES and MS system, since a proper system migration from an ES to MS system is an incremental process in which the most prominent components are extracted and remodularized as MSs first [1]. Such remodularized MSs run in a cloud setting and are integrated with the 'backend' enterprise system as depicted in the Fig. 1.

The software structure of an ES (e.g., an ERP system) consists of a set of self-contained modules (e.g., software components) drawn from different subsystems (e.g., production management), deployed on a specific execution platform.

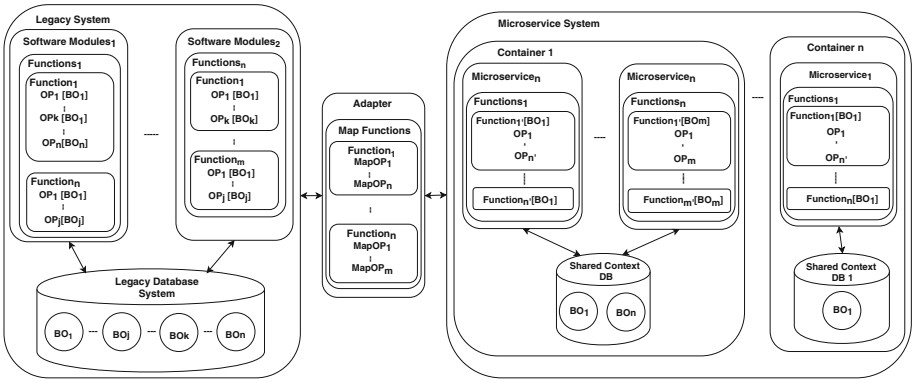

Fig. 1. Architecture of enterprise and microservices systems.

Modules consist of a set of functions (e.g., software classes) and each function consists of a number of operations (e.g., methods) aimed at manipulating BOs through CRUD operations which typically have database access logic or data processing logic applied to data stored in program variables and constants. The data stored in a centralized database associated with a deployed ES relates to BOs which process data resulting from business process executions supported through functions (such as transactions).

MSs are remodularized and potentially extended parts or functions of ESs, supporting consumer applications running in cloud applications. Since MSs are functionally isolated and loosely-coupled parts connected to each other, much like the components of a distributed system, they tend to concern individual BOs, locally managed through a database. The managed data of MSs is synchronized at discrete intervals with similar MS instances and with the backend ES. One or more MSs can run in an execution environment known as a cloud container, configured for specific execution characteristics, such as scalability or availability applying to all the MSs of the container.

Despite structural differences, the behaviour of an ES and an MS system is based on the invocation of operations, in well-defined processing sequences reflecting the relationships of BOs they manipulate. For example, the creation of a 'purchase order' will result in the invocation of functions involving the creation of 'line items' reflecting a strict containment of objects. Similarly, processing sequences between 'Shipper' and 'Shipping order' reflect weak containment while processing sequences between 'leads' and 'campaigns' reflect an association. In addition, normalization of a BO can result in additional process sequences. For example, the creation of a 'shipment' BO will result in an invocation of a function related to different shipments subtypes such as 'ground home delivery shipment' and 'intra UAESO shipment' based on the operational parameters provided at run time. These different execution sequences of operations reflect a set of single-entry-single-exit (SESE) regions [9] in an ES's executions.

Although an MS system executes in a manner similar to an ES, there are specific characteristics applicable only to an MS. Generally MSs run as distributed systems which are deployed through different containers that help the MS system to achieve high scalability and availability while executing services in an asynchronous manner and managing security through configurations of API gateways [1]. Scalability can be defined as allocation and de-allocation of the resources to containers on demand according to the configuration properties. Such configuration properties include load balancing and resource allocation policies optimizing the resources within a container allowing it to provide high scalability at lower cost. Furthermore, the configuration properties define circuit breaker threshold values for each MS which resides in a container, which assures that a request is redirected to another MS if it did not get a response from the initially accessed MS within the threshold time period resulting in high availability. Since there are multiple MSs in a single container, it can process multiple client requests in an asynchronous manner while aligning with the system requirements of the ES. All the MSs communicate through adapters which synchronize with the database system which resides in the legacy ES and this helps to achieve consistency among all the microservices which are distributed among multiple containers. Finally, each MS is developed in order to provide a specific functionality to the end user or the system, which makes them highly cohesive and loosely coupled services. This understanding leads us to the following formal characterization of the environment.

Let $\mathbb{I}$ and $\mathbb{O}$ be a universe of *input* types and *output* types, respectively. Let $\mathbb{OP}$, $\mathbb{T}$ and $\mathbb{B}$ be, respectively, a universe of *operations*, *database tables*, and *business objects*. Finally, let β be a binary relation on $\mathbb{B}$ such that β^+ is irreflexive[1]. Relation β defines a *subtype relation* on business objects, i.e., for every $(b_1, b_2) \in \beta^+$ we say that b_2 is a *subtype* of b_1. As proposed in this paper, techniques for the discovery of microservices rely on *abstractions* of ESs, as defined below.

The data related BOs in ESs are disseminated through several database tables.

Definition 2.1 (Business Object)
A *business object* b is characterized by a collection of database tables, i.e., $b \subseteq \mathbb{T}$. ⌟

The BOs in ESs have complex relationships with the operations which perform CRUD processes on them. Such operations are encapsulated in different functions of ESs and MSs.

Definition 2.2 (Operation)
An *operation op* is a triple (I, O, T), where $I \in \mathbb{I}^*$ is a sequence of *inputs*, $O \in \mathbb{O}^*$ is a sequence of *outputs*, and $T \subseteq \mathbb{T}$ is a set of *database tables*.[2] ⌟

[1] Given a binary relation α, by α^+ we denote the transitive closure of α.
[2] Given a set A, by A^* we denote the set of all finite sequences that can be generated by concatenating elements of A.

An ES can be seen as a finite automaton with operations as labels.

Definition 2.3 (Enterprise system).
An *enterprise system* is a 5-tuple $(Q, \Lambda, \delta, q_0, A)$, where:

- Q is a finite nonempty set of *states*,
- Λ is a set of *operations*, such that Q and Λ are disjoint,
- $\delta : Q \times (\Lambda \cup \{\tau\}) \to \mathcal{P}(Q)$ is the *transition function*, where τ a is a special *silent operation* such that $\tau \notin Q \cup \Lambda$,
- $q_0 \in Q$ is the *start state*, and
- $A \subseteq Q$ is the *set of accept states*.[3] ⌟

Let $\mathbb{C}$ and $\mathbb{M}$ be a universe of *containers* and *microservices*, respectively. Let S be an enterprise system. By $SESE(S)$, we denote the set of all (generalized) SESE fragments of S, cf. [9]; clearly, one can interpret an ES as a workflow graph with vertices defined by its states and a flow relation defined by its transition function. Each SESE fragment of an ES induces a *function*, or a *call graph*, i.e., a subgraph of ES. We abstract a function as a triple (I, O, OP), where I and O are sequences of inputs and outputs, respectively, and OP is a set of operations. For our purposes, we define a MSs system as follows.

Definition 2.4 (Microservices System). A *microservices system* of an enterprise system $S = (Q, \Lambda, \delta, q_0, A)$ is a 5-tuple (S, C, M, σ, μ), where:

- $C \subseteq \mathbb{C}$ is a set of *containers*,
- $M \subseteq \mathbb{M}$ is a set of *microservices*,
- $\sigma : C \to \mathcal{P}(M) \setminus \emptyset$ is a *deployment function* that maps each container $c \in C$ onto a non-empty set of microservices $\sigma(c)$ that are deployed on c, and
- $\mu : M \to \mathcal{P}(SESE(S)) \setminus \emptyset$ is a *microservice definition function* that maps each microservice $m \in M$ onto a non-empty set of SESE fragments, a.k.a *functions*, $\mu(m)$ of S, such that:
  - No two microservices are defined using the same function, i.e., $\forall\, m_1 \in M\, \forall\, m_2 \in M : (m_1 \neq m_2) \Rightarrow ((\mu(m_1) \cap \mu(m_2)) = \emptyset)$, and
  - Every two functions used to define the microservices in M are either disjoint, i.e., do not share an edge, or are in a subgraph relation. ⌟

Given an enterprise system S, $(S, \{c\}, \{m\}, \{(c, \{m\})\}, \{(m, \{S\})\})$, where $c \in \mathbb{C}$ and $m \in \mathbb{M}$, is its *elementary microservices system*, or the elementary enterprise and microservices architecture induced by S.

3 Automated Microservice Discovery

As described in Sect. 2, the behaviour of an ES and an MS system is based on the invocation of functions which consist of well-defined sequences of operations governed by BO relationships. Such sequences illustrate a particular execution pattern based on the structure and behaviour of an organization. Therefore, we argue that a proper analysis of these sequences of operations will help to

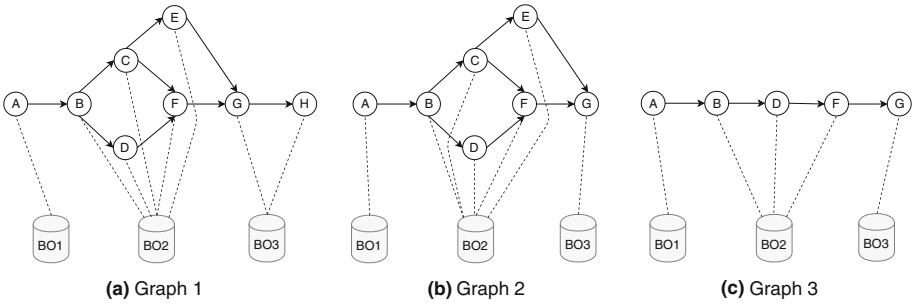

Fig. 2. Patterns of system executions and BO relationships.

derive prominent microserviceable components. This assumption leads us to two heuristics which assist in MS discovery.

As an example, assume that an ES has three hypothetical processing sequences, as depicted in Fig. 2, in which each node of the sequences represents a system state after performing a CRUD operation. These states are linked to the BOs on which different CRUD operations were performed. Figure 2(a) and (b) capture the same execution order dependencies between states 'A', 'B', 'C', 'D', 'E', 'F', and 'G'. Furthermore, the overlap between the execution patterns is high, i.e., more than 80%, which emphasizes that CRUD operations were performed on the same BOs. For instance, the campaign management module in SugarCRM describes different types of publicity campaigns, such as newsletter, email, and non-email. The execution paths and the BOs they execute upon are similar. However, the BO attributes they use in the execution processes are often different. This execution behaviour is explicit because of the structural splitting of objects at the BO level, as described by Halpin and Morgan [10]. To address this phenomenon, we define Heuristic 1.

Heuristic 1 ***(Subtype).*** *Given an enterprise system S, a subtype relation exists between a parent call graph $x = (I, O, OP) \in SESE(S)$ and a child call graph $x' = (I', O', OP') \in SESE(S)$, iff $I' \subseteq I$, $OP' \subseteq OP$, and $B' \subseteq B$, where B' and B are the BOs manipulated by OP' and OP, respectively. To ensure that the call graphs execute on the same BOs, we require that 80% of the states of the parent appear in the child.*

In addition, some execution sequences can occur often when executing a software system. As an example, the execution pattern 'A', 'B', 'D', 'F', 'G' occurs in Fig. 2(a), (b), and (c). This phenomenon depends on the functional relationships that occur during execution time. For example, 'B' precedes 'D' in every execution because, for instance, the data in 'B' is required for the execution of 'D'. In the functional structure level this can be described as a 'has a relationship' property, in which a class object of 'D' is referenced inside 'B'. Such

[3] Given a set A, by $\mathcal{P}(A)$, we denote the powerset of A.

functional structure emphasizes that the same behaviour should be preserved in all the system executions. To address this issue, we define Heuristic 2.

Heuristic 2 ***(Common Subgraph).*** *Given an enterprise system S, a common subgraph of two call graphs $x, x' \in SESE(S)$ is a call graph $x'' \in SESE(S)$, such that $x'' \subseteq x$ and $x'' \subseteq x'$.*

A common subgraph which captures frequent executions can be used as a basis for defining a microservice. This heuristic can be generalized to subgraphs common to multiple call graphs. Intuitively, choosing smaller common subgraphs produces smaller microservices which helps to achieve higher scalability. On the other hand, choosing larger subgraphs produces larger microservices which helps reduce communication overheads and improve system efficiency.

Heuristics 1 and 2 can guide the discovery of microservices that potentially support multiple microservice characteristics, such as high cohesion, low coupling, high scalability, availability, and processing efficiency, while preserving coherent features of enterprise systems. In what follows, this claim gets verified.

3.1 Discovery Process

Our microservice discovery and recommendation process based upon the above heuristics consists of two components, i.e., a *Business Object Analyser (BOA)* and a *System Dynamic Analyser (SDA)*, as depicted in Fig. 3.

Since MSs are focused around accessing and transferring states of BOs, or partitions of BOs, in the system [11], it is important to identify the BOs in a given ES. Therefore, the BOA is comprised of a System Operation Extraction Model (SOEM) and a Business Object Derivation Model (BODM). The SOEM evaluates all the SQL queries to identify the relationships between database tables, while the BODM derives the BOs based on the identified relationships and data similarities, as described by Nooijen *et al.* [12].

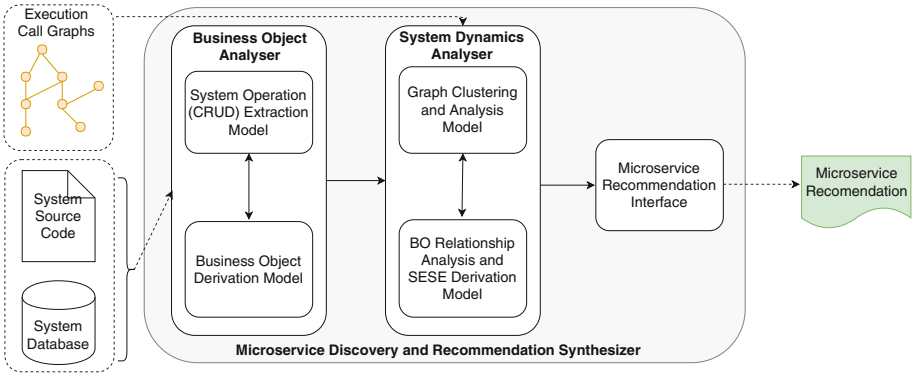

Fig. 3. An overview of our microservice discovery and recommendation process.

The BOs identified by the BOA are provided as input to the SDA along with call graphs of the ES. The graph clustering and analysis model in the SDA identifies the Frequent Execution Patterns (FEP) in the provided set of $SESE(S)$. These FEPs get evaluated against the aforementioned heuristics and classified into different categories, as described in Sect. 3.2. The categorized patterns are evaluated by BO relationship analysis and a SESE derivation model. The SESE derivation model identifies the BOs which are related to each node in the extracted graph pattern and the SESE regions related to each BO. Finally, the Microservice Recommendation Interface (MRI), provides different configuration models for MSs by evaluating the results of the system dynamic analysis model. Due to space limits, this paper only addresses the SDA and the MRI, which analyse the system execution patterns and recommend MS configuration models.

3.2 Microservice Discovery Algorithms

Given a set of call graphs of a legacy system, like the ones shown in Fig. 2, the SDA and MRI derive sets of MS recommendations based on Heuristics 1 and 2 using Algorithms 1 and 2. Algorithm 1 derives a set of subgraphs in the given set of call graphs of an ES, while Algorithm 2 analyses the subgraphs to identify functions which operate on single BOs to provide MS migration recommendations.

Algorithm 1 comprises four steps. The first step involves function $GRAPHSUMMARY$, which computes the set of adjacency matrices MT of the call graphs $SESE(S)$ (line 1). Each adjacency matrix is generated in two steps. First, the function constructs the set of all distinct states of the graphs. For example, for the three call graphs in Fig. 2, this set comprises states 'A', 'B', 'C', 'D', 'E', 'F', 'G', and 'H'. Then, for each call graph, the function creates a matrix $mt \in MT$ of size $N \times N$, where N is the number of distinct states (for the graphs in Fig. 2, the number of distinct states is eight). In a matrix $mt \in MT$, a transition between two states q and q' of the corresponding call graph is represented by '1' and the absence of a transition is represented by '0'.

The second step of Algorithm 1 constructs two matrices, the Augmented Adjacency Matrix (AAM) mt^a and Augmented Graph Matrix (AGM) mt^g, which are of size $N \times N$ (lines 3–10). At the beginning, all the values of mt^a are initialized to '0' and all the values of mt^g are initialized to the empty set. Then, the algorithm iterates over adjacency matrices $mt_k \in MT$ to compute statistics on transitions. The indices of the graphs that contain a transition are recorded in matrix mt^g and the number of graphs that contain the transition gets stored in matrix mt^a. The AAM and AGM generated for the call graphs in Fig. 2 are shown in Fig. 4. In Fig. 4(a), the value in the AAM row 'A', column 'B' is '3' because all the three call graphs depicted in Fig. 2 have a transition (an edge) from node 'A' to node 'B'. Similarly, in Fig. 4(b), the value in the AGM row 'A', column 'B' encodes the graphs that contain the corresponding transition. Since the transition is in all the three call graphs, the value has been set to '1', '2', '3'.

Algorithm 1. Calculate AAM and AGM

Require: An enterprise system S.
1: $MT = \{mt_1, \ldots, mt_n\} := GRAPHSUMMARY(SESE(S))$ *//Generate the summary matrix*
2: */* Iterate through each mt_k in MT */*
3: **for** each $k \in [1 \,..\, n]$ **do**
4: **for** each $i \in [0 \,..\, N-1]$, where N is the number of distinct states in $SESE(S)$ **do**
5: **for** each $j \in [0 \,..\, N-1]$ **do**
6: $mt^a[i][j] := mt^a[i][j] + mt_k[i][j]$
7: $mt^g[i][j] := mt^g[i][j] \cup \{k\}$
8: **end for**
9: **end for**
10: **end for**
11: $Sub = \langle sub_0, \ldots, sub_m \rangle := IDENTIFYSUBGRAPHS(mt^a, mt^g)$ *//Get common subgraphs*
12: **for** each $i \in [0 \,..\, m]$ **do**
13: $parents := \{mt \in MT \mid sub_i \text{ is a subgraph of } mt\}$
14: */* Record the similarity value for subgraph sub_i in the Sim list */*
15: $Sim_i := similarity(sub_i, parents)$
16: **end for**
17: **return** (Sub, Sim)

In the third step of Algorithm 1 the generated matrices mt^a and mt^g are passed as input to the *IDENTIFYSUBGRAPHS* function which computes the adjacency matrices of the common subgraphs *Sub* of the call graphs (line 11).

	A	B	C	D	E	F	G	H
A	0	3	0	0	0	0	0	0
B	0	0	2	3	0	0	0	0
C	0	0	0	0	2	2	0	0
D	0	0	0	0	0	3	0	0
E	0	0	0	0	0	0	2	0
F	0	0	0	0	0	0	3	0
G	0	0	0	0	0	0	0	1
H	0	0	0	0	0	0	0	0

(a) AAM

	A	B	C	D	E	F	G	H
A	Ø	1,2,3	Ø	Ø	Ø	Ø	Ø	Ø
B	Ø	Ø	1,2	1,2,3	Ø	Ø	Ø	Ø
C	Ø	Ø	Ø	0	1,2	1,2	Ø	Ø
D	Ø	Ø	Ø	Ø	Ø	1,2,3	Ø	Ø
E	Ø	Ø	Ø	Ø	Ø	Ø	1,2	Ø
F	Ø	Ø	Ø	Ø	Ø	Ø	1,2,3	Ø
G	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1
H	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

(b) AGM

Fig. 4. Intermediate matrices used by Algorithm 1 computed for the call graphs in Fig. 2.

In the fourth step, the algorithm iterates over subgraphs in *Sub* and measures the similarity Sim_i between subgraph sub_i and all its parent graphs (lines 12–16). The similarity is identified as the ratio of the number of nodes in sub_i to the number of distinct nodes in all the parent graphs. Finally, the algorithm returns the identified common subgraphs and calculated similarity values (line 17). If a similarity value is greater than 0.8 for a particular subgraph, we identify that the subgraph and its parent call graphs satisfy Heuristic 1.

The subgraphs which are common to all the call graphs in $SESE(S)$ satisfy Heuristic 2. However, further processing is required to identify functions that act upon single BOs. To accomplish this check, we present Algorithm 2.

Algorithm 2 consists of three steps. The first step involves identifying the states of the input subgraphs with no more than two incident transitions, a.k.a

Algorithm 2. Compute functions for given BOs

Require: A set of BOs B and list of graphs $Sub = \langle sub_0, \ldots, sub_m \rangle$
1: **for** each $i \in [0 \,..\, m]$ **do**
2: $Q := \emptyset$
3: **for** each node q in sub_i **do**
4: **if** q is incident to less than three edges **then**
5: $Q := Q \cup \{q\}$
6: **end if**
7: **end for**
8: $QS_i := Q$ *//QS is a list of sets of nodes*
9: **end for**
10: $Y := GENERATEGRAPHS(QS, Sub)$
11: $Z := Y$
12: **for** each $y \in Y$ **do**
13: /* Evaluate graph y to confirm that every operation in y is connected to the same $b \in B$ and if not remove it from Z */
14: **if** y operates on more than one BO in B **then**
15: $Z := Z \setminus \{y\}$
16: **end if**
17: **end for**
18: **return** Z *//Each function in Z operates over a single BO*

single-entry-single-exit states (lines 1–9). It is expected that the input subgraphs are generated by Algorithm 1. The loop of lines 1–9 iterates over all the subgraphs, while the loop of lines 3–7 runs over all the nodes of a current subgraph to extract and record the "SESE" states (line 5). The constructed sets of "SESE" states get stored in list QS on line 8.

In the second step of the algorithm function *GENERATEGRAPHS* constructs connected graphs composed of the nodes in QS that are subgraphs of the graphs in Sub, and records the result in set Y (line 10).

In the third step, the algorithm evaluates each graph $y \in Y$ to verify whether all the operations captured in y are carried out on the same BO (lines 11–17). If the operations relate to more than one BO, the graph gets removed from set Z, which initially is assigned to contain all the graphs in Y. The BO mapping is achieved by evaluating each database table t associated with operations of graph y and mapping t to the BOs that are characterized by t. If an operation, or several operations, of y relates to database tables that characterize more than one BO, then y gets removed from Z. At the end of the third step of Algorithm 2, set Z is composed of all the functions that operate on a single BO, and this set is returned on line 18. Finally, the functions in set Z get recommended to the user as possible MSs.

4 Implementation and Validation

A proper MS should provide high execution efficiency with a desirable level of scalability and availability. Furthermore the packages and components in it should be highly cohesive and loosely coupled [1,16]. In order to validate our MS discovery and recommendation process provides MSs with the desirable characteristics, we developed a prototype[4] based on the algorithm presented in Sect. 3.2

[4] https://github.com/AnuruddhaDeAlwis/Subtype.git.

and experimented on SugarCRM[5] and ChurchCRM[6] which is detailed in our technical report [17].

This section only presents the details of the experiments that we conducted using the prototype on SugarCRM, which is a customer relationship management system that has a complex system structure with more than 8,000 source files, 600 attributes divided between 101 tables. We specifically focused on the campaign management module of SugarCRM to generate the execution sequences for our microservice discovery process. In order to cover all the user cases related to the campaign management module, 10 different executions[7] related to the campaign management, such as target creation, campaign creation, and template creation, were performed and their log data was generated using the SugarCRM system's log functionality. The logs were then analyzed using the process mining tool Disco[8] and 10 different call graphs were generated, all together containing around 200 unique execution nodes. The generated call graphs and the database tables were provided as the input to the prototype.

Discovered MSs: Based on the call graphs and database tables, the prototype identified three subtypes of campaigns, namely newsletter, email, and non-email, which results in functional splitting of the ES based on object subtypes (satisfying Heuristic 1). In addition, the prototype identified common sequences related to all the executions resulting in functional splitting of the ES based on execution fragments (satisfying Heuristic 2).

Validation Process: The validation process was conducted by implementing the recommended MSs in Google Cloud. Each MS was hosted in Google Cloud using a cluster of size 2 which has two virtual CPUs and a total memory of 7.5 GB. The hosted MSs were exposed through the Google Cloud kubernetes API, allowing third party computers to access them via API calls. In order to validate the sub-typing recommendations, we implemented three MSs simulating newsletter, email, and non-email campaigns, and another system to simulate the legacy campaign module which covered all the campaign sub-types. In addition, we implemented a MS with common segments, i.e., fragments with similar states, communicating with other MSs simulating the common subgraphs recommendations given by the framework. Each MS was tested against a load of 150,000 requests and 300,000 requests generated by 10 machines simultaneously, simulating the customer requests, while recording their total execution time, average memory consumption and average disk consumption. The results are shown in Tables 1 and 2.

Based on the results reported in Tables 1 and 2, we calculated the scalability, availability, and execution efficiency of different combinations and the results obtained are summarized in Tables 3 and 4. The scalability was calculated according to the resources usage over time as described by Tsai *et al.* [13].

[5] https://www.sugarcrm.com/.
[6] http://churchcrm.io/.
[7] http://support.sugarcrm.com/Documentation/Sugar_Versions/8.0/Pro/Application_Guide/.
[8] https://fluxicon.com/disco/.

In order to determine the availability, first we calculated the time taken to process 100 requests if a particular MS is not available. Then, we used the difference between the total up-time and total down-time as described by Bauer *et al.* [14]. Efficiency gain was calculated by dividing the time taken by the legacy system to process all requests by the time taken by each MS. Furthermore we calculated the structural cohesion and coupling of the packages in legacy system and the new MS systems as described by Candela *et al.* [4].

Table 1. Legacy system vs subtype MSs execution results.

Campaign type	No. of requests	Ex. time (ms)	Avg mem (GB)	Avg disk (GB)
Legacy	150,000	324,000	3.00375	2.09550
Legacy	300,000	741,600	3.04025	2.10050
Newsletter	150,000	201,600	2.95475	2.09150
Newsletter	300,000	396,000	3.00575	2.09975
Email	150,000	198,000	2.89075	2.09225
Email	300,000	446,400	2.97075	2.10125
Non-email	150,000	226,800	2.84550	2.09300
Non-email	300,000	432,000	2.92875	2.10125

Table 2. Legacy system vs common subgraphs MSs execution results.

System type	No. of requests	Ex. time (ms)	Avg mem (GB)	Avg disk (GB)
Legacy	150,000	399,600	3.0335	2.0915
Legacy	300,000	781,200	3.1665	2.1020
Common Seg.	150,000	194,400	2.9110	2.0926
Common Seg.	300,000	396,000	2.9905	2.1015

Table 3. Scalability, availability, and efficiency gains using subtyping.

Campaign type	Scalability [Mem]	Scalability [Disk]	Availability [150,000]	Availability [300,000]	Efficiency [150,000]	Efficiency [300,000]
Legacy	2.652	2.626	99.856	99.918	1.000	1.000
Newsletter	1.963	1.937	99.910	99.956	1.607	1.873
Email	2.612	2.552	99.912	99.950	1.636	1.661
Non-email	1.867	1.821	99.899	99.952	1.429	1.717

Table 4. Scalability, availability, and efficiency gains using common subgraphs.

Campaign type	Scalability [Mem]	Scalability [Disk]	Availability [150,000]	Availability [300,000]	Efficiency [150,000]	Efficiency [300,000]
Legacy	1.9947	1.9205	99.9334	99.9667	1.0000	1.0000
Common MS	2.1314	2.0839	99.9334	99.9667	2.0556	1.9727

Table 5. Comparison of lack of cohesion and structural coupling.

System type	Lack of cohesion	Structural coupling
Legacy with campaign packages	104.00	17.00
MS with campaign packages	92.00	15.89
Legacy with commonality packages	55.83	18.00
MS with commonality packages	50.67	18.50

Experimental Results: According to Tsai *et al.* [13], the lower the number the better the scalability. Thus, the newsletter and non-email MSs have better scalability than the legacy system when considering both memory and disk usage over time (refer to Table 3). In the email MS there is a scalability gain, even thought it is not as significant as that of the gain in the newsletter and non-email MSs. When considering availability we clearly observe that there is higher availability in subtype MSs than in the legacy system. As the number of requests increased from 150,000 to 300,000, subtype MSs were able to handle the request overload while providing better availability than the legacy system. Most importantly, when examining the request processing efficiency, each subtyping MS managed to process the request at at-least 1.5 times the speed of the legacy system.

Table 4 reports that there is not much of a gain in scalability and availability in the MS discovered and developed based on Heuristic 2 when compared with the legacy system. In contrast, when comparing the efficiency gain, it is evident that the common MS managed to process requests at at-least twice the accelerated speed of the legacy system. Furthermore, when comparing the coupling and cohesion values detailed in Table 5, it is evident that both campaign and common MSs attained a higher level of cohesion than the legacy system. In addition, the campaign MS managed to achieve slightly better coupling when compared with the legacy system even though there is a small increase in coupling in the common MS. Similar results were obtained for the experiments conducted on ChurchCRM's service management module [17].

Provided Solutions: The obtained results have affirmed that MSs extracted based on the recommendation of our prototype can provide the same services to the users while preserving overall system behaviour and achieving higher scalability, availability, efficiency, high cohesion, and low coupling.

5 Related Work

Microservices have emerged as the latest style of service-based software allowing systems to be distributed through the cloud as fine-grained components, typically with individual operations, in contrast to services under SOA which include all logically related operations [1]. Even though microservices can support the evolution of ERP systems by providing exploitation in cloud-enabled platforms such as the IoT [2], the research conducted in this particular area is limited. To the best of our knowledge there is no research related to the automation of MS discovery in legacy systems, apart from the manual migrations achieved by Balalaie *et al.* [15]. Balalaie *et al.* have described the complexity associated with the system reengineering process while pointing out the importance of considering BOs and their relationships in the migration system process. Martin Fowler emphasizes the importance of adapting BO relationships in microservices [16] aligning with the Domain Driven Design principles.

However, the existing software re-engineering techniques do not consider the complex relationship of BOs with their behaviours in the re-engineering process. Furthermore, studies show that the success rate of existing software re-modularisation techniques, especially for large systems, remains low [4]. A key stumbling block is the limited insights available from syntactic structures of software code for profiling software dependencies and evaluating their measurements for coupling and cohesion metrics [5]. As such, to derive successful re-engineering techniques, a methodology should consider the enriched semantic insights available through the BOs and functions in an ES.

In such a process, the first challenge would be identifying the BOs which are distributed among several database tables in an ES system, and identifying the relationships between them. Nooijen *et al.* [12] and Lu *et al.* [8] proposed methodologies and heuristics to identify BOs based on the database schema and information in database tables. However, according to Lu *et al.*, the derived BOs might not be perfect and they have to be reclustered with the help of human expertise. A proper identification of BO relationships should consider the behavioural aspects of the systems as described by Hull [11]. However, there is still a gap in the area of correlating such behaviour with the underline BOs. As such, it is important to establish novel methodologies which incorporate both system behaviours and the business objects in the software re-engineering process.

6 Conclusion

This paper presented two heuristics used for functional splitting of ESs based on object subtypes and common execution fragments, while providing ground rules for MS discovery. A prototype was developed based on the proposed heuristics and validation was conducted by implementing the MSs recommended by the prototype for SugarCRM and ChurchCRM. The study has demonstrated that analysis of functions and BO CRUD operations while evaluating BO relationships helps to identify efficient solutions to migrate legacy systems into MSs

with high cohesion and low coupling while achieving better scalability, availability, and execution efficiency. However, further analysis of BO relationships, such as inclusive and exclusive containment should be considered to further optimize the MS discovery process, and this will be carried out as future work.

References

1. Newman, S.: Building MSs NGINX. O'Reilly (2015)
2. Columbus L.: Internet Of Things (IoT) Intelligence Update (2017). https://www.forbes.com/sites/louiscolumbus/2017/11/12/2017-internet-of-things-iot-intelligence-update/43aa6f4c7f31
3. Magal, S.R., Word, J.: Integrated Business Processes with ERP Systems. Wiley, Hoboken (2011)
4. Candela, I., Bavota, G., Russo, B., Oliveto, R.: Using cohesion and coupling for software remodularization: is it enough? ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**(3), 24 (2016)
5. Anquetil, N., Laval, J.: Legacy software restructuring: analyzing a concrete case. In: 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 279–286. IEEE, March 2011
6. Barros, A., Decker, G., Dumas, M., Weber, F.: Correlation patterns in service-oriented architectures. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 245–259. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71289-3_20
7. Pérez Castillo, R., García Rodríguez de Guzmán, I., Caballero, I., Piattini, M.: Software modernization by recovering web services from legacy databases. J. Softw. Evol. Process. **25**(5), 507–533 (2013)
8. Lu, X., Nagelkerke, M., van de Wiel, D., Fahland, D.: Discovering interacting artifacts from ERP systems. IEEE Trans. Serv. Comput. **8**(6), 861–873 (2015)
9. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 25–41. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19589-1_2
10. Halpin, T., Morgan, T.: Information modeling and relational databases. Morgan Kaufmann, Burlington (2010)
11. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88873-4_17
12. Nooijen, E.H.J., van Dongen, B.F., Fahland, D.: Automatic discovery of data-centric and artifact-centric processes. In: La Rosa, M., Soffer, P. (eds.) BPM 2012. LNBIP, vol. 132, pp. 316–327. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36285-9_36
13. Tsai, W.T., Huang, Y., Shao, Q.: Testing the scalability of SaaS applications. In: 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–4. IEEE, December 2011
14. Bauer, E., Adams, R.: Reliability and Availability of Cloud Computing. Wiley, Hoboken (2012)

15. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15
16. Fowler, M.: Microservices a definition of this new architectural term (2014). https://martinfowler.com/articles/microservices.html
17. De Alwis, A., Barros, A., Polyvyanyy, A., Fidge, C.: Technical report: function-splitting heuristics for discovery of microservices in enterprise systems (2018). https://eprints.qut.edu.au/119030/