Remodularization Analysis for Microservice Discovery Using Syntactic and Semantic Clustering

Adambarage Anuruddha Chathuranga De Alwis[1][0000-0002-4954-6595], Alistair Barros[1][0000-0001-8980-6841], Colin Fidge[1][0000-0002-9410-7217], and Artem Polyvyanyy[2][0000-0002-7672-1643]

[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. This paper addresses the challenge of automated remodularization of large systems as microservices. It focuses on the analysis of enterprise systems, which are widely used in corporate sectors and are notoriously large, monolithic and challenging to manually decouple because they manage asynchronous, user-driven business processes and business objects (BOs) having complex structural relationships. The technique presented leverages semantic knowledge of enterprise systems, i.e., BO structure, together with syntactic knowledge of the code, i.e., classes and interactions as part of static profiling and clustering. On a semantic level, BOs derived from databases form the basis for prospective clustering of classes as modules, while on a syntactic level, structural and interaction details of classes provide further insights for module dependencies and clustering, based on K-Means clustering and optimization. Our integrated techniques are validated using two open source enterprise customer relationship management systems, SugarCRM and ChurchCRM. The results demonstrate improved feasibility of remodularizing enterprise systems (inclusive of coded BOs and classes) as microservices. Furthermore, the recommended microservices, integrated with 'backend' enterprise systems, demonstrate improvements in key non-functional characteristics, namely high execution efficiency, scalability and availability.

Keywords: microservice discovery, system remodularization, cloud migration.

1 Introduction

Microservice architecture (MSA) has emerged as an evolution of service-oriented architecture (SOA) to enable effective execution of software applications in Cloud, Internet-of-Things and other distributed platforms [1]. Microservices (MSs) are fine-grained, in comparison to classical SOA components. They entail low coupling (inter-module dependency) and highly cohesive (intra-module dependency) functionality, down to individualised operations, e.g., single operation video-download as a MS component, versus a multi-operation video management SOA component [2]. This promotes systems performance properties, such as high processing efficiency, scalability and availability.

Reported experiences on MS development concern "greenfield" developments [1], where MSs are developed from "scratch". However, major uncertainty exists as to how MSs can be created by decoupling and reusing parts of a larger system, through refactoring. This is of critical importance for the corporate sectors which rely on large-scale

enterprise systems (ESs), (e.g., Enterprise Resource Planning (ERP) and Customer-Relationship Management (CRM)), to manage their operations. Analysing ESs and identifying suitable parts for decoupling is technically cumbersome, given the millions of lines of code, thousands of database tables and extensive functional dependencies of their implementations. In particular, ESs manage business objects (BOs) [3], which have complex relationships and support highly asynchronous and typically user-driven processes [4–6]. For example, an order-to-cash process in SAP ERP has multiple sales orders, having deliveries shared across different customers, with shared containers in transportation carriers, and with multiple invoices and payments, which could be processed before or after delivery [7]. This poses challenges to identify suitable and efficient MSs from ES codes using classical software refactoring and optimal splitting/merging of code across software modules.

Software remodularization techniques [8–10] have been proposed based on static analysis, to identify key characteristics and dependencies of modules, and abstract these using graph formalisms. New modules are recommended using clustering algorithms and coupling and cohesion metrics. The focus of static analysis techniques includes inter-module structure (class inheritance hierarchies), i.e., *structural inheritance relationships*, and inter-module interactions (class object references), i.e., *structural interaction relationships*. Given that a degradation of logical design reflected in software implementations can result in classes with low cohesion, other techniques have been proposed to compare structural properties of classes using information retrieval techniques [10], i.e., *structural class similarity*. Despite these proposals, studies show that the success rate of software remodularisation remains low [11].

This paper presents a novel development of software remodularization applied to the contemporary challenge of discovering fine-grained MSs from an ES's code. It extends the syntactic focus of software remodularization, by exploiting the semantic structure of ESs, i.e., BOs and their relationships, which are, in principle, influential in class cohesion and coupling. Specifically, the paper presents the following:

- A novel MS discovery method for ESs combining syntactic properties, derived from extracted *structural inheritance relationships*, *structural interaction relationships*, *structural class similarity*, and *semantic properties*, derived in turn from databases and the relationships of BOs managed by classes.
- An evaluation of the MS discovery methods that addresses three research questions (refer Section 4.1) by implementing a prototype and experimenting on two open-source CRMs: SugarCRM[3] and ChurchCRM[4]. The results show that there is a 26.46% and 2.29% improvement in cohesion and a 18.75% and 16.74% reduction in coupling between modules of SugarCRM and ChurchCRM, respectively. Furthermore, SugarCRM and ChurchCRM manage to achieve 3.7% and 31.6% improved system execution efficiency and 36.2% and 47.8% scalability improvement, respectively, when MSs are introduced to the system as suggested by our approach while preserving overall system availability (refer to Tables 1–6).

The remainder of the paper is structured as follows. Section 2 describes the related works and background on system remodularization techniques. Section 3 provides a

[3] https://www.sugarcrm.com/ [4] http://churchcrm.io/

detailed description of our MS discovery approach while Section 4 describes the implementation and evaluation. The paper concludes with Section 5.

2 Background and Motivation

This section first provides an overview of existing software remodularization and MS discovery techniques with their relative strengths and weaknesses. It then provides an overview of the architectural context of ESs and their alignments with MSs. This context is assumed in the presentation of our software remodularization techniques (Section 3).

2.1 Related Work and Techniques Used for Software Remodularization

Software remodularization techniques involve automated analysis of different facets of systems, including software structure, behaviour, functional requirements, and non-functional requirements. Techniques have focussed on static analysis to analyse code structure and database schemas of the software systems while dynamic analysis studies interactions of systems. Both approaches provide complementary information for assessing properties of system modules based on high cohesion and low coupling, and make recommendations for improved modularity. However, static analysis is preferable for broader units of analysis (i.e., systems or subsystems level) as all cases of systems' execution are covered compared to dynamic analysis [9].

Traditionally, research into software remodularization based on static analysis has focused on a system's implementation through two areas of coupling and cohesion evaluation. The first is structural coupling and cohesion, which focuses on structural relationships between classes in the same module or in different modules. These include *structural inheritance relationships* between classes and *structural interaction relationships* resulting when one class creates another class and uses an object reference to invoke its methods [8]. Structural relationships such as these are automatically profiled through Module Dependency Graphs (MDG), capturing classes as nodes and structural relationships as edges [8, 9], and are used to cluster classes using K-means, Hill-climbing, NSGA II and other clustering algorithms. The second is *structural class similarity* (otherwise known as *conceptual similarity* of the classes) [10]. This draws from information retrieval (IR) techniques, for source code comparison of classes, under the assumption that similarly named variables, methods, object references, tables and attributes in database query statements, etc., infer conceptual similarity of classes. Relevant terms are extracted from the classes and used for latent semantic indexing and cosine comparison to calculate the similarity value between them. Class similarity, thus, provides intra-module measurements for evaluating coupling and cohesion, in contrast to the inter-module measurements applied through structural coupling and cohesion described above.

Despite many proposals for automated analysis of systems, studies show that the success rate of software remodularization remains low [11]. A prevailing problem is the limited insights available from purely syntactic structures of software code to derive structural and interactional relationships of modules. More recently, semantic insights available through BO relationships were exploited to improve the feasibility of architectural analysis of applications. ESs manage domain-specific information using BOs, through their databases and business processes [5]. Evaluating the BO relationships

and deriving valuable insights from them to remodularize software systems falls under the category of *semantic structural relationships* analysis. Such semantic relationships are highlighted by the experiments conducted by Pĕrez-Castillo *et al.* [12], in which the transitive closure of strong BO dependencies derived from databases was used to recommend software function hierarchies, and by the experiments conducted by Lu *et al.* [13], in which SAP ERP logs were used to demonstrate process discovery based on BOs. Research conducted by De Alwis *et al.* [14,15] on MS discovery based on BO relationship evaluation shows the impact of considering semantic structural relationships in software remodularization. However, to date, techniques related to semantic structural relationships have not been integrated with syntactic structural relationships and structural class similarity techniques. As a result, currently proposed design recommendation tools provide insufficient insights for software remodularization.

2.2 Architecture for Enterprise System to Microservice Remodularization

As detailed in Section 2.1, there are multiple factors which should be considered in the MS derivation process. In this section, we define the importance of considering such factors with respect to the architectural configuration of the ES and MSs.

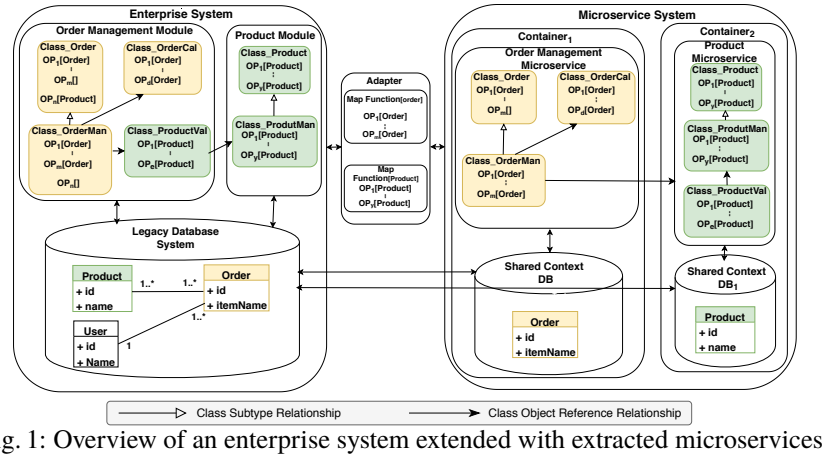

Fig. 1: Overview of an enterprise system extended with extracted microservices.

As depicted in Figure 1, an ES consists of a set of self-contained modules drawn from different subsystems and is deployed on a "backend" platform. Modules consist of a set of software classes which contain internal system operations and operations which manage one or more BOs through create, read, update, and delete (CRUD) operations. For example 'Order Management Module' consists of several classes such as 'Class_Order', 'Class_OrderCal' and 'Class_OrderMan', which contain operations manipulating data related to 'Order' BO and 'Class_ProductVal', which contain operations manipulating data related to 'Product' BO. Furthermore, the modules are interrelated through method calls between classes in different modules (see the relationship of 'Class_ProductVal' and 'Class_ProductMan' in Figure 1). In addition, classes inside each individual module can have generalization/specialization relationships (i.e., subtype-supertype relationships) between different classes as depicted by the relationships between 'Class_Order' and 'Class_OrderMan', and 'Class_Product' and 'Class_ProductMan' in Figure 1.

The MSs, on the other hand, support a subset of operations through classes which are related to individual BOs. Such implementations lead to high cohesion within MSs and low coupling between the MSs (see the 'Order Management Microservice' and 'Product Microservice' in Figure 1). The MSs communicate with each other through API calls in case they require information related to different BOs which reside in other MSs. For example, 'Order Management Microservice' can acquire Product values through an API call to 'Product Microservice' (refer arrow between the MSs in Figure 1). The execution of operations across the ES and MS system is coordinated through business processes, which means that invocations of BO operations on the MSs will trigger operations on ES functions involving the same BOs. As required for consistency in an MS system, BO data will be synchronised across databases managed by ES and MSs periodically.

Based on this understanding of the structure of the ES and MSs, it is clear why we should consider semantic and syntactic information for the MS discovery process. In order to capture the *subtype relationships* and *object reference relationships* that exist in the ES system, we need *structural inheritance relationship* and *structural interaction relationship* analysis methods. Such methods can help to group classes which are highly coupled into one group, such as the grouping of 'Class_Order', 'Class_OrderCal' and 'Class_OrderMan' into one 'Order Management Microservice', as depicted in Figure 1. However, those relationships alone would not help to capture class similarities at the code level. For example, the 'Class_ProductVal' operates on 'Product' BO and relates to the 'Product Module' much more than the 'Order Management Module'. Such information can be captured using *structural class similarity* measuring methods. With *structural inheritance and interaction relationships* and *structural class similarity* we can cluster classes into different modules. However, such modules might not align with the domain they are related to until we consider the BO relationships of different classes. In Figure 1, one can notice that different classes in the ES relate to different BOs. As such, it is of utmost importance to consider the *semantic structural relationships* in the MS derivation process, since each MS should aim to contain classes that are related to each other and perform operations on the same BO (refer to the 'Order Management Microservice' and 'Product Microservice' in Figure 1).

Previous research has extensively used *structural relationships* in system remodularization [8–10]. However, when it comes to MS derivation, combining the *semantic structural relationships* with the *syntactic structural relationships* should allow deriving better class clusters suitable for MS implementation. Given this system architecture context and our understanding of the features that should be evaluated for MS systems, we developed algorithms, as described in Section 3, for MS discovery. We use the following formalisation here onwards to describe the algorithms.

Let $\mathbb{I}$ and $\mathbb{O}$ be a universe of *input* types and *output* types, respectively. Moreover, let $\mathbb{OP}$, $\mathbb{B}$, $\mathbb{T}$ and $\mathbb{A}$ be, respectively, a universe of *operations*, *BOs*, *database tables* and *attributes*. We characterize a *database table* $t \in \mathbb{T}$ by a collection of attributes, i.e., $t \subseteq \mathbb{A}$, while a *business object* $b \in \mathbb{B}$ is defined as a collection of database tables, i.e., $b \subseteq \mathbb{T}$. An *operation op*, either of an ES or MS system, is given as a triple (I, O, T), where $I \in \mathbb{I}^*$ is a sequence of *input types* the operation expects for input, $O \in \mathbb{O}^*$ is a sequence of *output types* the operation produces as output, and $T \subseteq \mathbb{T}$ is a set of *database*

tables the operation accesses, i.e., either reads or augments.[5] Each *class cls* $\in$ *CLS* is defined as a collection of operations, i.e., *cls* $\subseteq \mathbb{OP}$.

3 Clustering Recommendation for Microservice Discovery

In order to derive the MSs while considering the factors defined in Section 2, we developed a six-step approach, which is illustrated in Figure 2. In the first step, we derive the BOs by evaluating the SQL queries in the source code structure and also the database schemas and data as described by Nooijen *et al.* [16]. Next, we analyse the semantic structural relationships by deriving the class and BO relationships. Steps 3–5 are used to discover the syntactic details related to the ES. In the third step, we measure the structural class similarities between the classes and in steps 4 and 5 we capture the structural details of the classes, step 4 discovers the structural inheritance relationships and step 5 discovers the structural interaction relationships. The details obtained through steps 2–5 are provided to the final step in which a K-means clustering algorithm is used to cluster and evaluate the best possible combination of classes for MS development and finally suggest them to the developers. Detailed descriptions of these steps and corresponding algorithms are provided in Section 3.1.

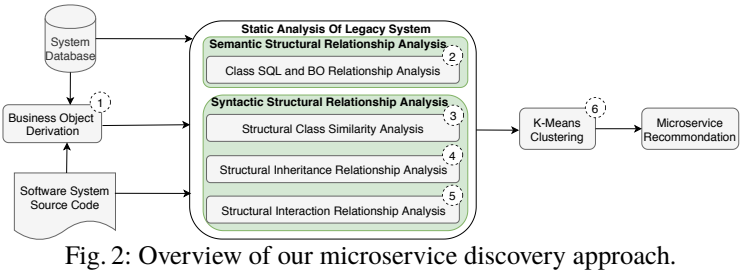

Fig. 2: Overview of our microservice discovery approach.

3.1 Clustering Discovery Algorithms

As depicted in Figure 2, in order to derive a satisfactory clustering of system classes and operations and suggest MSs recommendations, we supply the K-means algorithm with four main feature sets. To derive these feature sets, we use Algorithm 1, which is composed of eight steps.

In an ES, the information related to a BO is often stored in several database tables. Thus, we define a BO $b \in B$ as a collection of database tables, i.e., $b \subseteq T$. As such, to identify the BOs, in the first step, the *BOS* function is performed by Algorithm 1, which derives the BOs, B, of the system through the analysis of the database table relationships and their data similarities, as described by Nooijen *et al.* [16] (see line 1). In the second step of the algorithm, the function *CLSEXT* is used to extract code related to each class *cls* $\in$ *CLS* from the system code by searching through its folder and package structure (see line 2).

In the third step, we extract information required for the structural class similarity analysis using information retrieval (IR) techniques. As such, in the third step, the algorithm identifies unique words *UW* related to all the classes using function *UWORDEXT*

[5] A^* denotes the application of the Kleene star operation to set A.

(see line 3) which requires all the source codes of the classes *CLS*, and stop words *STW*, which should be filtered out from the classes. In general, IR techniques analyse documents and filter out the content which does not provide any valuable information for document analysis, which are referred to as 'stop words'. In our case, the stop words (*STW*) contain syntax related to the classes, common technical terms used in coding in that particular language (in this case PHP) and also common English words which would not provide any valuable insight about class purpose. These are specified by the user based on the language of the system they evaluate. The function *UWORDEXT* first filters out the stop words *STW* from the classes *CLS* and then identifies the collection of unique words *UW* in classes *CLS*, which is generally referred to as a 'bag of words' [17].

Algorithm 1: Discovery of BO and class relationships

Input: System code *SC* of an ES *s*, stop words related to classes *STW* and system database *DB*

Output: Feature set data *borel*, *cosine*, *subtyperel*, *referencerel* and BOs *B*

```
1  B = {b_1, ..., b_n} := BOS(SC, DB)
2  CLS = {cls_1, ..., cls_m} := CLSEXT(SC)
3  UW = ⟨uw_1, ..., uw_z⟩ := UWORDEXT(CLS, STW)
4  for each cls_i ∈ CLS do
5  |   for each b_k ∈ B do
6  |   |   borel[i][k] := BCOUNT(cls_i, b_k);
7  |   end
8  |   for each uw_s ∈ UW do
9  |   |   uwcount[i][s] := WCOUNT(uw_s, cls_i);
10 |   end
11 end
12 for each cls_i, cls_k ∈ CLS do
13 |   cosine[i][k] := COSINECAL(uwcount[i], uwcount[k]);
14 end
15 subtyperel := SUBTYPECAL(CLS);
16 referencerel := REFERENCECAL(CLS);
17 return borel, cosine, subtyperel, referencerel, B
```

In the fourth step, the algorithm evaluates each class ($cls \in CLS$) extracted in step two and identifies the BOs which are related to each class. For this purpose, the algorithm uses the function *BCOUNT* that processes the SQL statements, comments and method names related to the classes and counts the number of times tables relate to BOs. This information is stored in matrix *borel* (see lines 5–7). In this matrix, each row represents a class, and each column represents the number of relationships that class has with the corresponding BO, as depicted in Figure 3(a). This helps to capture the semantic structural relationships (i.e., BO relationships) data, which provides an idea about the "boundness" of classes to BOs. For example Class 1 'Cls 1' is related to 'BO1' and 'BO2' in Figure 3(a).

In the fifth step, the algorithm derives another matrix *uwcount*, which keeps the count of unique words related to each class using the function *WCOUNT* (see lines 8–10). In this matrix, again, rows correspond to classes, and columns correspond to

unique words identified in step three of the algorithm that appear in the corresponding classes. The values in *uwcount* are then used in the sixth step to calculate the cosine similarity between the documents using *COSINECAL* function (see lines 12–14). First, this function normalizes the term frequencies with the respective magnitude L2 norms. Then it calculates the cosine similarity between different documents, by calculating the cosine value between two vectors of the *uwcount* (i.e., the rows related to two classes in *uwcount* matrix) and stores the values in the *cosine* matrix, as exemplified in Figure 3(b). Note that the cosine similarity of a class to itself is always '1'. This provides the structural class similarity data for clustering.

Next, we extract the structural inheritance relationships (i.e., the class subtype relationships) and structural interaction relationships (i.e., the class object reference relationships). This is achieved through steps seven and eight in the algorithm which use function *SUBTYPECAL* (see line 15) to identify the subtype relationships and function *REFERENCECAL* (see line 16) to identify the class object reference relationships. In both of these functions, as the first step, the code is evaluated using Mondrian[6], which generates graphs based on class relationships. Then, the graphs are analyzed to create two matrices, namely *subtyperel* and *referencerel* which, respectively summarize the class subtype and reference relationships for further processing (see *subtyperel* depicted in Figure 3(c) and *referencerel* depicted in Figure 3(d)).

	BO1	BO2	BO3
Cls 1	1	3	0
Cls 2	0	3	0
Cls 3	0	2	0
Cls 4	1	0	3
Cls 5	4	0	0

(a) borel

	Cls 1	Cls 2	Cls 3	Cls 4	Cls 5
Cls 1	1	0.75	0.83	0.44	0.05
Cls 2	0.75	1	0.65	0.51	0.02
Cls 3	0.83	0.65	1	0.53	0.03
Cls 4	0.44	0.51	0.53	1	0.12
Cls 5	0.05	0.02	0.03	0.12	1

(b) cosine

	Cls 1	Cls 2	Cls 3	Cls 4	Cls 5
Cls 1	1	1	1	0	0
Cls 2	1	1	0	1	0
Cls 3	1	0	1	1	0
Cls 4	0	1	1	1	0
Cls 5	0	0	0	0	1

(c) subtyperel

	Cls 1	Cls 2	Cls 3	Cls 4	Cls 5
Cls 1	1	0	0	0	0
Cls 2	0	1	1	1	0
Cls 3	0	1	1	1	1
Cls 4	0	1	1	1	0
Cls 5	0	0	1	0	1

(d) referencerel

Fig. 3: Matrices derived from Algorithm 1.

The feature set data *borel*, *cosine*, *subtyperel*, *referencerel* and BOs *B* obtained from Algorithm 1 are provided as input to the K-Means algorithm (i.e., Algorithm 2) to cluster the classes related to BOs based on their syntactic and semantic relationships. Note that each dataset captures different aspects of relationships between classes in the given system (see Figure 3). Each initial centroid $intcent \in IntCent$ is a row number in the dataset that we provide. For example, one can select the first row of the dataset (as we have done in Figure 3, see highlighted in red), as an initial centroid. In that situation, the *IntCent* will contain the data related to that specific row of the data set. Given these datasets as the first step in Algorithm 2, we initialize the distance difference value *distDif* to some constant, e.g., 10. The *distDif* is responsible for capturing the distance differences between the initial centroids *IntCent* and the newly calculated centroids *NewCent*. If this distance difference is zero, then it means that there is no difference between the initial centroid values and the newly calculated centroid values (in which case the algorithm terminates). After initializing the *distDif* value, the next steps of the algorithm are performed iteratively until the aforementioned condition of *distDif* is met (see lines 2–21).

The first step of the iterative execution is to initialize the set of clusters *CLUS*, which we use to store the node groups identified by Algorithm 2. Next, we need to identify the cluster that each row (or the node) of our data should belong to by comparing

[6] https://github.com/Trismegiste/Mondrian

the distance between each node in the dataset and each node in the initial centroids *intcent* ∈ *IntCent*. Hence we iterate through each row of the dataset we obtained from Algorithm 1 (see line 4 in Algorithm 2), while calculating the Euclidean distance between each row and each initial centroid *intcent* ∈ *IntCent* (see lines 4–12 in Algorithm 2). For this calculation, as the initial step, we define the minimum Euclidean distance value *minEquclidianDis* and initialize it to *MAX_INTEGER* (e.g., 100000). We assign this value to the *minEquclidianDis* to ensure that it would be larger than the value we obtain for the *newEuclideanDis* (line 7) at the end of the first iteration. Then, we calculate the Euclidean distance between one data set, for example, row 1 in Figure 3 and each initial centroid point given. Next, we identify the centroid which has the minimum Euclidian distance to the node we obtained and allocate that node number to that particular cluster *clus* ∈ *CLUS* (line 13). This process is carried out until all the nodes are clustered based on the Euclidean distance calculation. In the end, each node in the data set is clustered towards the centroid which has the minimal distance to it based on the four feature sets which emphasize that the classes related to that particular cluster are bound to the same BO and to each other syntactically and semantically.

Algorithm 2: K-Means clustering for microservice discovery

Input: *borel*, *cosine*, *subtyperel*, *referencerel*, k which is the number of BOs B and an array of initial Centroid values *IntCent*

Output: *CLUS* which captures the clustered MS recommendations.

```
1  distDif := 10 ;                                          // initialize distDif value
2  while distDif ≠ 0 do
3      CLUS = {clus_1, ..., clus_k} := INITCLUSTERS(k);
4      for 0 ≤ i < borel.size() do
5          minEuclideanDis := MAX_INTEGER;                  // initialize minEuclideanDis
6          for each intcent_j ∈ IntCent do
7              newEuclideanDis := EUCAL(intcent_j, borel[i], cosine[i],
                 subtyperel[i], referencerel[i]);
8              if newEuclideanDis < minEuclideanDis then
9                  minEuclideanDis := newEuclideanDis;
10                 clusterNumber := j;
11             end
12         end
13         clus_clusterNumber := clus_clusterNumber + i;
14     end
15     for each clus_i ∈ CLUS do
16         NewCent = {newcent_1, ..., newcent_n} := NEWCENTCAL(clus_i);
17     end
18     distDif := DISTANCECAL(IntCent, NewCent);            // Calculate distances
19     IntCent := NewCent;
20 end
21 return CLUS
```

The next step of Algorithm 2 is to calculate the new centroids based on the clusters obtained. For this, we take the mean value of the node data sets belonging to each

cluster and assign it as the new centroid (see function *NEWCENTCAL* at lines 15–17 in Algorithm 2). Then, we calculate the distance difference between the initial centroids and the new centroids. If this difference is zero, it means that there is no change of the centroid points and the algorithm has come to the optimum clustering point. If not, the newly calculated centroids becomes the initial centroids for the next iteration of the algorithm. At the end of the algorithm, the final set of clusters which contain the classes of the analysed ES are provided to the developers as recommendations for constructing MSs based on them.

4 Implementation and Validation

To demonstrate the applicability of the method described in Section 3, we developed a prototypical MS recommendation system[7] capable of discovering the class clusters related to different BOs, which lead to different MS configurations. The system was tested against two open-source customer relationship management systems: SugarCRM and ChurchCRM. SugarCRM consists of more than 8,000 files and 600 attributes in 101 tables, while ChurchCRM consists of more than 4,000 files and 350 attributes in 55 tables. However, most of the files are HTML files which are related to third-party components used by the systems. For the clustering, we only used the 1,400 classes of SugarCRM and 280 classes of ChurchCRM which capture the core functionality of the systems. Using our implementation, we performed static analysis of the source code to identify the BOs managed by the systems. As a result, 18 BOs were identified in SugarCRM, e.g., account, campaign, and user, and 11 BOs in ChurchCRM, e.g., user, family, and email. Then, we performed static analysis of both systems to derive matrices, similar to those depicted in Figure 3, summarizing the BO relationships, class similarity relationships, class subtype relationships and class object reference relationships. All the obtained results were processed by the prototype to identify the class clusters to recommend MSs. Based on the input, the prototype identified 18 class clusters related to the BOs in SugarCRM and 11 class clusters related to the BOs in ChurchCRM. Consequently, each cluster suggests classes for developing an MS that relates to a single BO.

4.1 Research Questions

Our evaluation aims to answer three research questions:

- **RQ1:** Do syntactic and semantic relationships between source code artifacts of an ES convey useful information for its remodularization into MSs?
- **RQ2:** Can our MS recommendation system discover MSs that have better cohesion and coupling than the original ES modules and lead to high scalability, availability, and execution efficiency in the cloud environment?
- **RQ3:** Can our MS recommendation system discover MSs that lead to better scalability, availability, and execution efficiency in the cloud environment than some MSs that do not follow the recommendations?

[7] https://github.com/AnuruddhaDeAlwis/KMeans.git

4.2 Experimental Setup

To answer the above research questions, we set up the following experiment consisting of three steps. In the first step, we evaluated the effectiveness of considering four different features (i.e., feature 1: *borel*, feature 2: *cosine*, feature 3: *referencerel* and feature 4: *subtyperel* extracted in Algorithm 1) in the clustering process. This was achieved through measuring the Lack of Cohesion (LOC) and Structural Coupling (StrC) of the clusters, as described by Candela *et al.* [11], while incrementally adding different features in the clustering process. We calculated the values for the ES by clustering the classes into folders while preserving the original package structure, see first rows in Tables 1–4. Then, we clustered the classes several times, each time adding more features and calculating the LOC and StrC values. The obtained values are reported in Tables 1–4.

After evaluating the effectiveness of various features for clustering, we assessed the efficacy of introducing MSs to the ES. To this end, first, we hosted each ES in an AWS cloud by creating two EC2 instances having two virtual CPUs and a total memory of 2GB, as depicted on the left side of Figure 4. The data related to the systems were stored in a MySQL relational database instance which has one virtual CPU and total storage of 20GB. These systems were then tested against 100 and 200 executions generated by four machines simultaneously, simulating the customer requests, while recording their total execution time, average CPU consumption, and average network bandwidth consumption (refer to our technical report [18]). For SugarCRM, we simulated the functionality related to *campaign creation*, while for ChurchCRM we simulated the functionality related to *adding new people* to the system. For the simulations, we used Selenuim[8] scripts which executed the system in a way similar to a real user.

Next, we introduced the 'campaign' and 'user' MSs to the SugarCRM system and 'person' and 'family' MSs to the ChurchCRM system. Each MS was hosted on an AWS elastic container service (ECS), which has two virtual CPUs and a total memory of 1GB, as depicted on the right side of Figure 4. The data related to the BOs of each MS (i.e., campaign BO and user BO data of SugarCRM and person BO and family BO data of ChurchCRM) was stored in separate MySQL relational database instances with one virtual CPU and total storage of 20GB. Next, the executions were performed on both ESs, again simulating *campaign creation* for SugarCRM and *adding new people* for ChurchCRM. Since MSs are extended parts of the ESs in these executions, the ESs used API calls to pass the data to the MSs and the MSs processed and sent back the data to the ESs. The data in the MS databases and ES databases were synchronized using the Amazon database migration service. Again, we recorded the total execution time, average CPU consumption, and average network bandwidth consumption for the entire system (i.e., ES and MS as a whole) (refer to our technical report [18]). Based on the attained values, we calculated the scalability, availability, and execution efficiency of the different combinations, and the obtained results are summarized in Tables 5–6 as *ES with MSs(1)* (refer to the second rows in Tables 5–6). Scalability was calculated according to the resource usage over time, as described by Tsai *et al.* [19]. To determine availability, first we calculated the packet loss for one minute when the system is down and then obtained the difference between the total up time and total time (i.e., up time + down time), as described by Bauer *et al.* [20]. Efficiency gain was calculated by dividing

[8] https://www.seleniumhq.org/

the total time taken by the legacy system to process all requests by the total time taken by the corresponding ES system which has MSs.

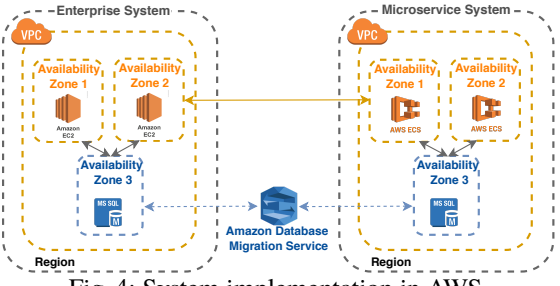

Fig. 4: System implementation in AWS.

In the third experiment, we disrupted the suggestions provided by our recommendation system and developed 'campaign' and 'user' MSs for SugarCRM, while introducing operations related to 'campaign' to 'user' MS and operations related to 'user' to 'campaign' MS. Similarly, for ChurchCRM, we developed 'person' and 'family' MSs such that 'person' MS contains operations related to 'family' MS and 'family' MS contains operations related to 'person' MS. With this change, again, we set up the experiment as described earlier and obtained the experimental results (refer to our technical report [18]). Then we calculated the scalability, availability and execution efficiencies of the systems which are summarized in Tables 5–6 as *ES with MSs(2)* (refer to the third rows in Tables 5–6). Based on these obtained experimental results we evaluate the effectiveness of the algorithms by answering the posed research questions.

RQ1: Impact of syntactic and semantic relationships. The lower the lack of cohesion and structural coupling numbers, the better the cohesion and coupling of the system [11]. Consequently, it is evident from the average numbers reported in Tables 1–4 (refer to the orange color cells) that clustering improved the cohesion of software modules of SugarCRM and ChurchCRM by 26.46% and 2.29%, respectively, while reducing the coupling between modules by 18.75% and 16.74% respectively. Furthermore, it is evident that introducing additional features (i.e., syntactic and semantic information) in the clustering process increased the number of modules which obtain better coupling and cohesion values (refer to the blue cells in Tables 1–4). Thus, we conclude that there is a positive effect of introducing multiple syntactic and semantic relationships to the clustering process to improve the overall performance of the system.

RQ2: Recommended MSs vs original ES. According to Tsai *et al.* [19], the lower the measured number, the better the scalability. Thus, it is evident that the MS systems derived based on our clustering algorithm managed to achieve 3.7% and 31.6% improved system execution efficiency and 36.2% and 47.8% scalability improvement (considering CPU scalability) (refer Tables 5 and 6), for SugarCRM and ChurchCRM, respectively, while also achieving better cohesion and coupling values (refer Tables 1–4). As such, our recommendation system discovers MSs that have better cohesion and coupling values than the original enterprise system modules and can achieve improved cloud capabilities such as high scalability, high availability and high execution efficiency.

Table 1: ChurchCRM ES vs MS System Lack of Cohesion Value comparison.

Features	1	2	3	4	5	6	7	8	9	10	11	Avg
Original ES	61	188	853	7	4	1065	31	378	3064	13	17	516.45
1 and 2	61	77	666	33	8	1453	73	351	3802	3	10	594.27
1,2 and 3	61	77	853	3	4	1564	23	351	3064	13	17	548.18
1,2,3 and 4	58	188	820	7	3	1059	31	351	3012	10	15	504.90

Table 2: ChurchCRM ES vs MS System Structural Coupling Value comparison.

Features	1	2	3	4	5	6	7	8	9	10	11	Avg
Original ES	41	26	61	17	16	70	29	31	123	27	19	41.81
1 and 2	41	25	8	37	20	64	33	31	121	3	7	35.45
1,2 and 3	41	25	61	16	16	68	29	27	123	7	19	41.09
1,2,3 and 4	42	25	34	17	15	63	29	3	112	26	17	34.81

Table 3: SugarCRM ES vs MS System Lack of Cohesion Value comparison.

Features	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Avg
Original ES	32	19	1255	698	1482	0	163	693	349	45	171	1803	1058	0	66	47	317	522	484.44
1 and 2	19	0	1067	1122	1173	0	86	459	170	21	120	953	587	0	36	6	453	187	358.83
1,2 and 3	19	0	1201	626	1173	0	86	459	170	45	120	1027	587	0	36	7	590	268	356.33
1,2,3 and 4	19	0	1201	626	1173	0	86	459	170	45	120	1027	587	0	36	5	590	268	356.22

Table 4: SugarCRM ES vs MS System Structural Coupling Value comparison.

Features Considered	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Avg
Original ES	24	12	121	63	48	4	99	24	29	29	28	101	67	0	16	10	82	85	46.77
1 and 2	12	2	117	76	48	4	87	18	25	20	27	85	45	0	15	5	52	64	39.00
1,2 and 3	12	2	116	53	48	4	87	18	22	29	27	79	45	0	15	5	41	83	38.11
1,2,3 and 4	12	2	116	53	48	4	87	18	22	29	27	79	45	0	15	3	41	83	38.00

Table 5: Legacy vs MS System EC2 characteristics comparison for SugarCRM.

System Type	Scalability [CPU]	Scalability [DB CPU]	Scalability Network	Availability [200]	Availability [400]	Efficiency [200]	Efficiency [400]
ES only	3.521	2.972	2.759	99.115	99.087	1.000	1.000
ES with MSs(1)	2.246	2.532	2.352	99.082	99.086	1.037	1.000
ES with MSs(2)	2.667	2.546	2.684	99.099	99.099	1.018	0.986

Table 6: Legacy vs MS System EC2 characteristics comparison for ChurchCRM.

System Type	Scalability [CPU]	Scalability [DB CPU]	Scalability Network	Availability [200]	Availability [400]	Efficiency [200]	Efficiency [400]
ES only	3.565	3.109	3.405	99.385	99.418	1.000	1.000
ES with MSs(1)	1.859	2.751	3.663	95.000	94.871	1.316	1.189
ES with MSs(2)	2.876	2.667	2.779	95.238	95.000	1.250	1.158

RQ3: Recommended MSs vs some MSs. MSs developed based on the suggestions provided by our recommendation system for SugarCRM and ChurchCRM managed to achieve: (i) 36.2% and 47.8% scalability improvement in EC2 instance CPU utilization, respectively; (ii) 14.8% and 11.5% scalability improvement in database instance CPU utilization, respectively; (iii) while achieving 3.7% and 31.6% improvement in execution efficiency, respectively. However, MSs that violate the recommendations reduced (i) EC2 instance CPU utilization to 24.24% and 19.32% ; (ii) execution efficiency to 1.8% and 2.5%, for SugarCRM and ChurchCRM respectively and reduced database instance CPU utilization to 14.3% for SugarCRM. As such, it is evident that the MSs developed by following the recommendations of our system provided better cloud characteristics than the MSs developed against these recommendations.

4.3 Limitations

Next, we discuss two important limitations of our approach.

Limitation of BO derivation: To derive the BOs related to the given ESs, we used the method described by Nooijen *et al.* [16]. However, as described by Lu *et al.* [13], this method cannot derive BOs correctly without validation from the system developers. Hence, in this paper, we manually evaluated accuracy of the derived BOs by referring to the systems' manuals and documentation.

Limitation of structural class similarity analysis: The structural class similarity analysis obtained a 'Bag of Words' term frequency and, finally, calculated the cosine similarity between the documents. The first limitation of this method is filtering out of valuable information in the data preprocessing stage. This issue was mitigated by manually evaluating the stop words used in the text preprocessing step. In addition, cosine values might not provide an accurate idea about the structural class similarity since the structural similarity may also depend on the terms used in the definitions of the class names, method names and descriptions given in comments. This was mitigated to a certain extent by evaluating the code structure of the software systems before evaluating and verifying that the class names, method names and comments provide valuable insights into the logic behind the classes that implement the system.

5 Conclusion

This paper presented a novel technique for automated analysis and remodularization of ESs as MSs by combining techniques which consider semantic knowledge, together with syntactic knowledge about the code of the systems. A prototype recommendation system was developed and validation was conducted by implementing the MSs recommended by the prototype for two open source ESs: SugarCRM and ChurchCRM. The experiment showed that the proposed technique managed to derive class clusters which would lead to MSs with desired Cloud characteristics, such as high cohesion, low coupling, high scalability, high availability, and processing efficiency. In future work, we will enhance the technique by considering method level relationships in the analysis of MS candidates.

Acknowledgment: *This work was supported in part, through the Australian Research Council Discovery Project: DP190100314, "Re-Engineering Enterprise Systems for Microservices in the Cloud".*

References

1. Newman, S., 2015. Building microservices. O'Reilly Media, Inc.
2. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
3. Barros, A., Duddy, K., Lawley, M., Milosevic, Z., Raymond, K. and Wood, A., 2000, October. Processes, roles, and events: UML concepts for enterprise architecture. In International Conference on the Unified Modeling Language (pp. 62-77). Springer, Berlin, Heidelberg.
4. Schneider, T., 2012. SAP Business ByDesign Studio: Application Development (pp. 24-28). Boston: Galileo Press.
5. Decker, G., Barros, A., Kraft, F.M. and Lohmann, N., 2008, December. Non-desynchronizable service choreographies. In International Conference on Service-Oriented Computing (pp. 331-346). Springer, Berlin, Heidelberg.
6. Barros, A., Decker, G. and Dumas, M., 2007, May. Multi-staged and multi-viewpoint service choreography modelling. In Proceedings of the Workshop on Software Engineering Methods for Service Oriented Architecture (SEMSOA), Hannover, Germany. CEUR Workshop Proceedings (Vol. 244).
7. Barros, A., Decker, G., Dumas, M. and Weber, F., 2007, March. Correlation patterns in service-oriented architectures. In International Conference on Fundamental Approaches to Software Engineering (pp. 245-259). Springer, Berlin, Heidelberg.
8. Praditwong, K., Harman, M. and Yao, X., 2010. Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering, 37(2), pp.264-282.
9. Mitchell, B.S. and Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering, 32(3), pp.193-208.
10. Poshyvanyk, D. and Marcus, A., 2006, September. The conceptual coupling metrics for object-oriented systems. In 22nd IEEE International Conference on Software Maintenance (pp. 469-478). IEEE.
11. Candela, I., Bavota, G., Russo, B. and Oliveto, R., 2016. Using cohesion and coupling for software remodularization: Is it enough?. ACM Transactions on Software Engineering and Methodology (TOSEM), 25(3), p.24.
12. PÃľrezâĂŘCastillo, R., GarcÃŋaâĂŘRodrÃŋguez de GuzmÃąn, I., Caballero, I. and Piattini, M., 2013. Software modernization by recovering web services from legacy databases. Journal of Software: Evolution and Process, 25(5), pp.507-533.
13. Lu, X., Nagelkerke, M., van de Wiel, D. and Fahland, D., 2015. Discovering interacting artifacts from ERP systems. IEEE Transactions on Services Computing, 8(6), pp.861-873.
14. De Alwis, A.A.C., Barros, A., Fidge, C. and Polyvyanyy, A., 2019, October. Business Object Centric Microservices Patterns. In OTM Confederated International Conferences: On the Move to Meaningful Internet Systems (pp. 476-495). Springer, Cham. (LNCS, volume 11877)
15. De Alwis, A.A.C., Barros, A., Polyvyanyy, A. and Fidge, C., 2018, November. Function-splitting heuristics for discovery of microservices in enterprise systems. In International Conference on Service-Oriented Computing (pp. 37-53). Springer, Cham. (LNCS, volume 11236).
16. Nooijen, E.H., van Dongen, B.F. and Fahland, D., 2012, September. Automatic discovery of data-centric and artifact-centric processes. In International Conference on Business Process Management (pp. 316-327). Springer, Berlin, Heidelberg.
17. Lebanon, G., Mao, Y. and Dillon, J., 2007. The locally weighted bag of words framework for document representation. Journal of Machine Learning Research, 8(Oct), pp.2405-2441.
18. https://drive.google.com/file/d/19niZYleVsuboNETCScYRB9LFVi3_5F2z/view?usp=sharing
19. Tsai, W.T., Huang, Y. and Shao, Q., 2011, December. Testing the scalability of SaaS applications. In 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 1-4). IEEE.
20. Bauer, E. and Adams, R., 2012. Reliability and availability of cloud computing. John Wiley & Sons.