Availability and Scalability Optimized Microservice Discovery from Enterprise Systems

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

[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. Microservices have been introduced to industry as a novel architectural design for software development in cloud-based applications. This development has increased interest in finding new methodologies to migrate existing enterprise systems into microservices to achieve desirable performance characteristics such as high scalability, high availability, high cohesion and low coupling. A key challenge in this context is discovering microserviceable components with promising characteristics from a complex monolithic code base while predicting their resulting characteristics. This paper presents a technique to support such re-engineering of an enterprise system based on the fundamental mechanisms for structuring its architecture, i.e., business objects managed by software functions and their interactions. The technique relies on queuing theory and business object relationship analysis. A prototype for microservice discovery and characteristic analysis was developed using the NSGA II software clustering and optimization technique and has been validated against two open-source enterprise systems, SugarCRM and ChurchCRM. Our experiments demonstrate that the proposed approach can recommend microservice design which improves scalability, availability and execution efficiency of the system while achieving high cohesion and low coupling in software modules.

Keywords: Microservice discovery, system reengineering, system optimization

1 Introduction

Microservices were introduced around 2011 to the software industry and interest in microservices has increased over the years due to the development and deployment advantages they provide over monolithic system architectures. A microservice architecture encourages development of applications as small independent services, each running its own process and while communicating with other microservices via REST API calls [1]. Even though different industry giants such as Netflix^{TM}, and now Twitter^{TM}, eBay^{TM} and Amazon^{TM} have adapted their systems to microservices, they have not been adopted for the dominant form of software in businesses, namely enterprise systems, limiting such systems' evolution and their ability to exploit the full benefits of modern cloud-enabled platforms such as Google Cloud, Amazon AWS and IoT [2].

Enterprise systems, such as enterprise resource planning (ERP) and customer relationship management (CRM), are large and complex and contain complex business processes encoded in application logic managing business objects, in typically many-to-many relationships [3]. Restructuring such a system into microservices is an error-prone task due to several reasons. Firstly, it is difficult to identify the highly cohesive and loosely coupled functions and operations which could be usefully separated as microservices, by a vast code base. Secondly, it is challenging to figure out an optimal splitting of the system functionalities as fine-grained microservices while minimizing the communication costs (i.e., service calls) between them. Thirdly, it is difficult to predict the system's scalability and availability behaviour based on the components identified as microservices without implementing them. The third issue is a major concern because implementing a system to validate its scalability and availability performance incurs additional cost and time, and the developers might need to conduct several implementations to validate the best configuration for the microservice system's development.

Automated software re-engineering techniques have been proposed to improve the efficiency of transforming legacy applications and structures [4], into a Service-Oriented-Architecture (SOA) using static analysis techniques (i.e., source code analysis) [5] and dynamic analysis techniques (i.e., execution log and pattern analysis) [6]. However, these techniques have, to date, not been applied to the re-engineering challenges of microservices. More specifically, there has been no research conducted in the area of deriving microservices from enterprise systems while evaluating the scalability and availability of the resulting microservices.

This paper presents discovery techniques that support the identification of suitable consumer-oriented parts of enterprise systems which could be re-engineered as microservices based on knowledge gained through business object relationships and their execution patterns while analysing their scalability and availability characteristics to provide better microservice configurations. A microservice recommendation process was developed using the Non-dominated Sorting Genetic Algorithm (NSGA) II and was validated against two open source customer management systems, SugarCRM[3] and ChurchCRM[4]. Our experiments showed that our methodology can be used to discover microservices which improve system structure and achieve high scalability, availability and execution efficiency of the system while achieving high cohesion and low coupling in software modules.

2 Related Work

Restructuring existing monolithic systems into new architectures has been an important branch in the software engineering research community. However, being a relatively new concept to the business-centric software industry, microservices have had very limited research conducted in the area of system re-engineering and restructuring. Even though there are some approaches for microservice discovery, a better understanding of monolithic to microservice migration can be obtained through the manual migration report of Balalaie *et al.* [7]. They describe the complexity associated with the system re-engineering process while pointing out the importance of considering business objects

[3] https://www.sugarcrm.com/

[4] http://churchcrm.io/

and their relationships in the system migration process. Further, Martin Fowler emphasizes the importance of adapting business object relationships in microservices [8] by mentioning Domain Driven Design (DDD) principles [9]. DDD specifically focuses on identifying business objects that are related to the same domain, which helps to develop software components that are highly cohesive and loosely coupled.

Even though there is research about discovering business objects in enterprise systems and analysing their complex relationships [10, 11], research related to re-engineering enterprise systems while considering the enriched semantic insights available through the complex relationship of business objects is limited. As described by Fuguo *et al.*, business objects in enterprise system play a major role in the overall system structure and their effect can even be seen at the API level of the system [12]. A proper evaluation of such relationships and identification of functions related to each business object leads to software components or microservices which align with the single responsibility principle [13], which makes the components highly cohesive and loosely coupled.

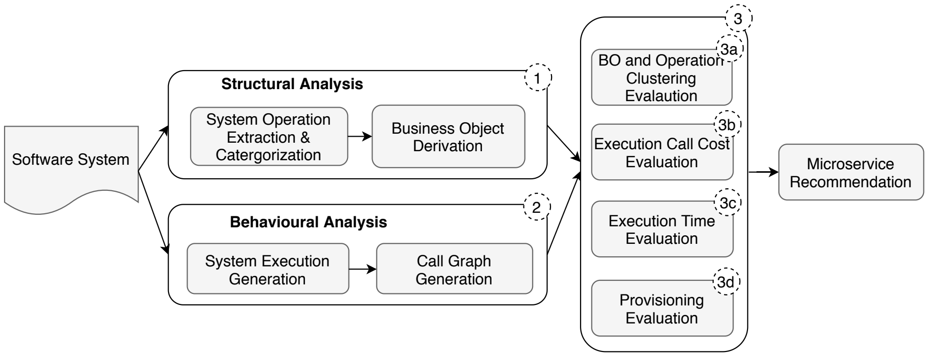
Apart from business object relationships, the number of execution calls between different microservices plays a major role in defining high performing microservices, because an excessive number of network calls can increase response times while decreasing the availability of the service [1]. Available research related to microservice discovery has considered the number of calls between different methods to suggest microservices for the developer by analysing system execution logs [14, 15]. However, such research has not considered the possibility of evaluating scalability and availability to derive better microservice configurations from enterprise systems. Even though there is queuing theory based research about evaluating system scalability [16] and ways of suggesting system configurations while evaluating system workload [17], applying such theories to microservice discovery has yet to be done, and is the focus of our research herein.

3 Microservice Discovery and Optimization Model

To discover microservices with desirable characteristics we developed a three-step approach, which is illustrated in Fig. 1. First we perform static analysis on the system in order to derive the business objects it manipulates. To achieve this, we extract and evaluate all the SQL queries in the given enterprise system's code and identify the relationships between database tables. These relationships are then used to derive the business objects according to the approach described by Nooijen *et al.* [10]. In the second step, a behavioural analysis is performed in order to generate and extract data related to system execution. For this we execute the system, simulating the users' behaviour with the help of Selenium scripts. These simulations generate system execution logs which are then used to generate call graphs related to the executions. Finally, as the third step, all the structural and behavioural details generated are provided to an optimization algorithm in order to discover a high performing system partitioning for microservices. The optimization criteria were derived by answering the four research questions below.

RQ1: *How can we derive highly cohesive components out of a given enterprise system which will provide better microservices?*

In the microservice literature, the bounded context related to the DDD is presented as a promising design rationale for identifying microservices. According to this rationale, each microservice should correspond to a single and defined bounded context, such that all the operations in the microservice should correspond to the changes in that particular context [1]. Generally, each bounded context can be defined as an artifact or a business object in an enterprise system [7]. As such, discovering the business objects in an enterprise system and extracting the operations performed on each business object leads to the discovery of microservices with a proper bounded context. In this situation, each microservice is changed only for a single reason (i.e., each microservice is aligned with the single responsibility) which leads to a highly cohesive microservice system. As such, our optimization algorithm should group business objects and operations related to (i.e., performed on) each business object into different clusters. To evaluate the optimization level of such clustering we implemented the BO and operation clustering evaluation (i.e., 3a in Fig. 1) as the third step in our process.



Software System
Structural Analysis
1
System Operation Extraction & Catergorization
Business Object Derivation
Behavioural Analysis
2
System Execution Generation
Call Graph Generation
3
3a
BO and Operation Clustering Evalaution
3b
Execution Call Cost Evaluation
3c
Execution Time Evaluation
3d
Provisioning Evaluation
Microservice Recommendation

Fig. 1. Overview of our microservice discovery approach.

RQ2: *What is the criterion needed to evaluate the coupling between two microservice components?*

Coupling can be defined as the dependency evaluation criterion between two classes, packages or modules [18]. Given three software packages 'A', 'B', 'C', if there is a higher number of calls between 'A' and 'C' than 'A' and 'B', then one can define that 'A' and 'C' are more tightly coupled than 'A' and 'B', because the number of interactions between 'A' and 'C' is higher than the number of interactions between 'A' and 'B'. In fact this is one of the criteria used in software re-modularisation when clustering software packages to achieve better coupling [5, 18]. If software packages are developed properly there should be low coupling between packages (i.e., a low number of inter-package calls) and high coupling between the classes in the same package (i.e., a high number of intra-package calls) [5, 18]. Similarly, when defining microservices, one should choose the level of operation clustering to minimize inter-microservice communication while maximizing intra-microservice communication. As such, to evaluate the coupling in microservice discovery we implemented an execution call cost evaluation (i.e., 3b in Fig. 1) step in our process.

RQ3: *What is availability of a software system and how can we measure it?*

Availability of a system can be defined in two perspectives, namely the probability that a system is operational at a given time and the probability of the system providing a response to the customer within a given time limit. The basic method to measure the system's operation time is to calculate the ratio between the service up time and the total time [19]. This measure provides an idea about the probability of service unavailability experienced by a customer. However, sometimes systems take more time than expected to provide a response to the customer even though the system is available. If the response time is too long then customers tend to leave the system [20]. Generally, the service up time of a given system is based on the particular environment and the execution situation. As such, it is difficult to predict such behaviour without an implementation. However, when it comes to the response time one can predict it based on the time it takes to transfer a message between two microservices and the time it takes to execute that particular message.

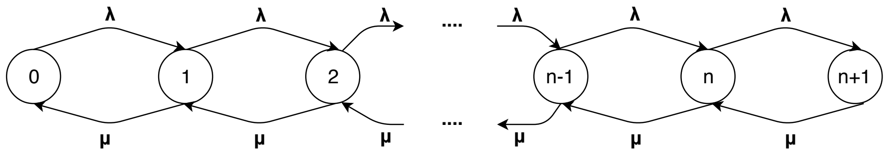


Fig. 2. Queuing model for a single microservice process.

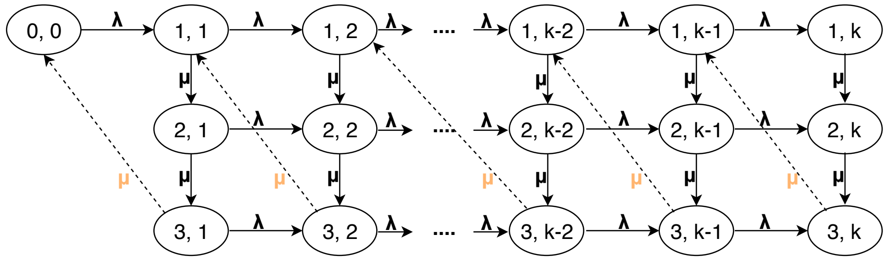


Fig. 3. Queuing model for a multiple microservices process.

In the literature, for different predictions related to system provisioning and performance, monitoring queuing theory has been used as reliable method [21, 22] and it has been clearly noted that such predictions can achieve solutions close to the corresponding real world scenarios [16]. In our work, we have considered two possible scenarios related to microservice execution. The first scenario would be where only a single microservice is processing customer requests. The second scenario would be the situation where there are multiple microservices in the system and they interact with each other to process a customer request. In Fig. 2 each circle with a number represents a microservice instance of the same microservice. As such, Fig. 2 showcases the scenario where multiple microservice instances will be created to support customer requests at a given time.

Similarly, in Fig. 3, the circles represent the microservice instances. In this situation the first comma-separated number inside the circle represents the microservice and the second number represents the instance of that microservice. For example, if we take '1,2', this represents the second instance of the first microservice. As such, in Fig. 3, we have depicted three microservices $(1, 2, 3)$ with their related instances. The first scenario can be defined using a birth and death process in queue modeling as depicted in Fig. 2 while the second one can be defined using a matrix as in Fig. 3. In both Figs. 2 and 3 variable λ is the request arrival rate and μ is the execution rate of the system.

In order to derive the models needed, we assumed that requests arrive at a Poisson rate but they get processed in an exponential rate and the system is in a steady state. Given such a model, one can define the response time for a particular customer request to be the total of the message transfer time (i.e., $1/\lambda$) and execution time (i.e., $1/\mu$). However, there are several other variables which should be counted in this process such as the microservice provisioning time, CPU usage, Memory usage, IO and network bandwidth. In this work, we assume that all the microservices have the same CPU, Memory and IO configurations. However, when consider the provisioning time, microservices take time to start up when there are multiple services residing in the same container [23]. As such, we add the provisioning time for each microservice to the response time calculation. Furthermore, we assume that there is enough bandwidth such that an increase in the number of requests will not reduce the message transformation speed between clients and microservices [23, 24]. Furthermore, we assume that the operational complexity of each process related to microservices is similar. These assumptions and models lead to the execution time evaluation (i.e., 3c in Fig. 1) step in our process.

RQ4: *What is scalability of a software system and how can we measure it?*

Scalability can be described as the ability of a cloud layer to increase its capacity by expanding its resource quantity (e.g., CPU and Memory) by consuming lower layer resources [26]. A more advanced concept of scalability would be elasticity which is a measure of resource provisioning and de-provisioning over time. According to Herbst *et al.*, one can calculate provisioning characteristics by monitoring the change in the amount of resources allocated and the time it takes to allocate those resources [27]. Generally, one can define the amount of memory required for request processing to be dependent on the data transferred into the system and the amount of data transferred within the system. As such, in order to derive the amount of memory allocated to each microservice, we use the number of inter-microservice and intra-microservice calls. The provisioning times are simply derived from the experimental results of Amaral *et al.* [23]. In this scenario we assume that the microservice allocates its total memory requirement in the provisioning time and until that the system is in an under-provisioned state. As such, the system which spends less time and uses fewer resources in an under-provisioned state provides better scalability. This leads to the provisioning evaluation (i.e., 3d in Fig. 1) step in our process.

A detailed overview of the algorithm used to implement the above four criteria is provided in Section 4.

4 NSGA II Optimization

In order to discover an optimal microservice configuration while evaluating the four criteria described in Section 3, we chose the Non-dominated Sorting Genetic Algorithm II (NSGA II) which is a multi-objective optimization algorithm which provides an optimal set of solutions while achieving global optima, when there are multiple conflicting objectives to be considered [28]. NSGA II can provide near optimal solutions when used to cluster software packages and classes to achieve high cohesion and low coupling [5].

Algorithm 1 provides microservice configuration solutions using three execution steps and requires the population size (n), number of generations (Gen), chromosome length (C_Len), crossover probability (Cr_Prob) and mutation probability (Mut_Prob) as input data. Apart from the above standard parameters, our algorithm requires further input, such as the BOs of the system (B), and execution graph nodes (N) and their relationships (R) extracted from the execution graphs. These details can be extracted from a software system based on the methodology described by De Alwis *et al.* [14]. The population size (n) defines how many chromosomes are populated in a single generation, while the number of generations (Gen) defines the number of times the algorithm generates different populations before it stops. The crossover probability (Cr_Prob) and mutation probability (Mut_Prob) are responsible for defining the probability of performing crossovers and mutations on chromosomes. Interested readers can find further details about NSGA II elsewhere [28].

Here onwards we describe our algorithm variant based on the hypothetical execution graph depicted in Figure 4. In the graph, each node illustrates an operation executing in the system and the 'BO's illustrate the business objects that each operation executes on.

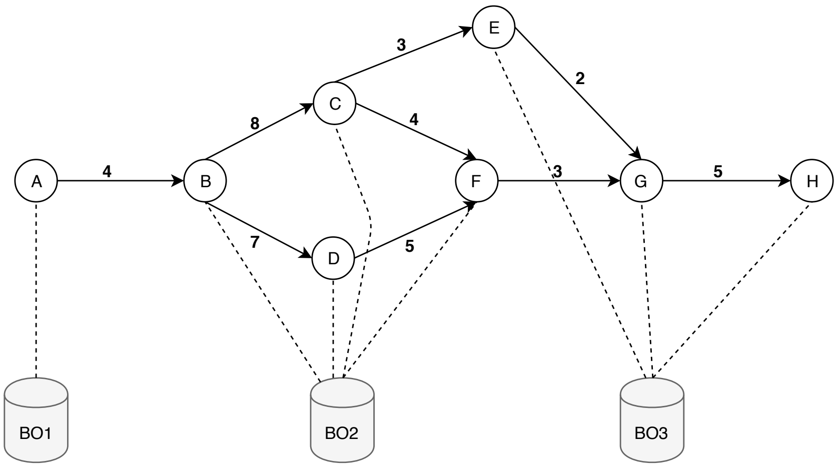


Fig. 4. Hypothetical execution graph of a system process.

The first step of the algorithm involves the *SYNPOP* function which synthesizes a parent population of the given size n (see line 1). Function *SYNPOP* uses a random number generator to generate chromosomes of length C_Len, where a chromosome is a sequence of numbers each representing a node in the execution graph. A chromosome generated for the execution graph in Fig. 4 can be represented as a sequence of numbers '0, 1, 2, 3, 4, 5, 6, 7', in which the numbers refer to the corresponding graph nodes 'A, B, C, D, E, F, G, H', such that 0 refers to A, 1 refers to B, etc. Apart from generating the parent population, *SYNPOP* calculates and stores the fitness for each parent. The fitness calculation is preformed in two steps. First, the algorithm calculates the maximum cost (Max_c) for a chromosome as $\sum_{i=0}^{C\_Len} 2^i$ which can achieve a highest value of 255 (i.e., 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128). Then the algorithm calculates the costs for the four criteria that we described in Section 3. Since we need to achieve high cohesion as our first objective, the algorithm should be able to cluster system operations with their related business objects. To calculate such cohesion the cost function should be able to represent to which extent the nodes related to the same BO have been grouped together. As such, for each chromosome, the clustering cost ($Cost_c$) is calculated as $Cost_c = \sum_{i=0}^{Clus} \sum_{j=0}^{d} 2^d$, where d is the distance from the first occurrence of a node related to a particular BO to the next occurrence of the node related to the same BO within the chromosome and *Clus* is the number of business object clusters. When considering the scenario given in Fig. 4 high cohesion can be achieved by clustering the operations into three groups containing nodes 'A', 'B, C, D, F' and 'E, G, H' based on the business objects they execute on. This leads to a total clustering cost of 23 (i.e., 1 + 15 + 7).

Similarly, to achieve low coupling (i.e., the second objective described in Section 3), the cost of execution calls ($Cost_e$) between clusters is computed as the sum of inter-cluster calls between the different clusters. For the running example, i.e., the chromosome '0, 1, 2, 3, 5, 4, 6, 7', this sum would be $4 + 3 + 3 = 10$, because the costs of calls between the pairs of clusters in '0', '1, 2, 3, 5', '4, 6, 7', are '4', '3' and '3'. Further details for calculating the first two objectives for the NSGA II algorithm can be found elsewhere [14].

Availability (i.e., the third objective described in Section 3) is dependent on the time it takes to provide a response to a customer. As such to derive a cost function to measure availability (*Ava*) we used data related to inter-cluster and the intra-cluster calls. For example, if we take the number of inter-cluster calls between '0', '1, 2, 3, 5', '4, 6, 7' it would be 4 and 6, and the number of intra-cluster calls would be 0, 24 and 7. We consider the number of inter-cluster calls as the number of requests sent to each microservice and the number of intra-cluster calls as the number of requests passed within the microservice. As such, we define the total amount of time needed to transfer the data to a microservice (i.e., λ in Figs. 2 and 3) as the number of inter-cluster calls ($Call_i$) multiplied by the data packet size ($Data_s$) divided by the bandwidth of the network (*Band*) (i.e., $(Call_i \times Data_s)/Band$). The execution time of a microservice (i.e., μ in Figs. 2 and 3) is calculated as the number of intra-cluster calls ($Call_e$) multiplied by the data packet size ($Data_s$) divided by the process complexity (*Com*) (i.e., $(Call_e \times Data_s)/Com$). In this calculation, we assumed that all the packets transferred have the same amount of data and each packet is of the maximum TCP packet size which is 64kb. Furthermore, we assumed that the bandwidth is equal to the general Ethernet bandwidth of 10Mbit/s. The time taken for internal data transfer is not taken into consideration because microservices have the same transfer speed as a native system which is negligible [29]. The process complexity is given a fixed value assuming that none of the operations are related to floating point executions [25]. Since provisioning time (*Pro*) also affects the response time of a microservice we add provisioning time values obtained from Amaral *et al.* [23] to the total response time. Since we evaluate two scenarios where only a single microservice responses to a customer request (as depicted in Fig. 2) and multiple microservices response to customer requests (as depicted in Fig. 3), two cost functions were defined and evaluated. For a single microservice with x number of operational instances we defined the cost function as $Ava = ((Call_i \times Data_s)/Band + (Call_e \times Data_s)/Com + Pro) \times x$. For a microservice system with y number of different microservices, in which each microservice has x number of operational instances we defined the cost function as $Ava = ((Call_i \times Data_s)/Band + (Call_e \times Data_s)/Com \times (y-1) + Pro) \times x$.

Algorithm 1: NSGA II Algorithm adapted for microservice discovery

```
Input: $n, Gen, C\_Len, Cr\_Prob, Mut\_Prob, B, N, R$
Output: A list of clustering of BOs and OPs for MSs
1  $Pop^p = \langle pop_1, \ldots, pop_n \rangle := SYNPOP(n, C\_Len, \gamma, B, N, R)$;
2  $Pop^c := Rank^f := \langle\rangle$;
   /* Perform crossover and mutation to generate child population */
3  while $Pop^c.length() < n$ do
4      if $RANDOM(0,1) < Cr\_Prob$ then
5          $Pop^c := CROSSOVER(Pop^p, Pop^c)$;
6      if $RANDOM(0,1) < Mut\_Prob$ then
7          $Pop^c := MUTATION(Pop^p, Pop^c)$;
8  end
9  for $each\ i \in [1..Gen]$ do
10     $Pop^t := Pop^p + Pop^c$;
11     $Rank^f = \langle rank_1^f, \ldots, rank_m^f \rangle := FNDS(Pop^t)$;
12     if $i = Gen$ then
13         break;
       /* Identify the Pareto front of the generated population and rank them */
14     $Pop^c := \langle\rangle$;
15     for $k \in [1..m]$ do
16         if $length(rank_k^f) < (n - length(Pop^c))$ then
17             $Pop^c := Pop^c + rank_k^f$;
18         else
19             $Pop^c := Pop^c + CCS(rank_k^f)$;
20     end
21     $Pop^p := Pop^c$;    // Initialize new parent population
22     $Pop^c = \langle pop_1^c, \ldots, pop_n^c \rangle := SYNCHD(Pop^p)$;
23 end
24 return $(Rank^f)$
```

As described in Section 3, to calculate scalability (*Scal*) we need to figure out the amount of resources used and the time taken to provision the resources. Here we assume that the total amount of memory required for each microservice is similar to the total number of packets it receives and processes. Thus we calculate the total memory requirement for a microservice as $(Call_i + Call_e) \times Data_s$ and the provisioning time values are obtained from Amaral *et al.* [23]. For a microservice system with x instances we defined the cost function as $Scal = (Call_i + Call_e) \times Data_s \times x$. The *fitness* for a given chromosome is finally obtained as $fitness = Max_c - (Cost_c + Cost_e + Ava + Scal)$.

The second step of the algorithm generates the child population by performing crossover operations and mutation operations on the parent chromosomes (see lines 3–8). In order to perform the crossover operation, the algorithm selects two parents using binary tournament selection [28]. This is performed by randomly identifying two parent chromosomes and extracting the chromosome with the highest fitness value out of them. After identifying two parent chromosomes for crossover, the algorithm splits the first parent chromosome from a predefined position (normally half of the chromosome's length) and includes it as the first part of the child chromosome. As the second part of the chromosome it includes the genes extracted from the second parent which are not in the first part of the child chromosome.

After generating the first child population, the algorithm generates *Gen* new populations, (refer to lines 9–23 in Algorithm 1) which constitutes the third (and last) step of the algorithm. First, the current total population Pop^t is computed at line 10 by concatenating the parent population Pop^p and the child population Pop^c. Next, the algorithm calculates the non-dominated fronts, or the Pareto fronts, of the total population. A non-dominated front contains the chromosomes which have the optimal values for the four objectives that were defined above, namely the clustering cost ($Cost_c$), cost of execution calls ($Cost_e$), availability cost (Ava) and scalability cost ($Scal$). The chromosome's optimization of node clustering is calculated as the difference between the maximum possible cost of the chromosome and cost of its node clustering (i.e., $Max_c - Cost_c$). Similarly, the chromosome's optimization of execution calls is calculated as the difference between the maximum possible cost of the chromosome and the cost between its cluster calls (i.e., $Max_c - Cost_e$). The chromosome's optimization of availability and scalability is similarly calculated by obtaining the difference between maximum possible cost of the chromosome and cost of scalability and availability (i.e., $Max_c - Ava$ and $Max_c - Scal$). The non-dominated chromosomes in Pop^t are extracted as the first front using function *FNDS* (see line 11). After extracting the first non-dominated front, the algorithm evaluates the other chromosomes in Pop^t and identifies the second non-dominated front. This process is repeated until all the chromosomes are categorised into different fronts (2, . . ., m), where each generated front may contain multiple non-dominated chromosomes.

Once the Pareto fronts are obtained, a new child population is created by concatenating the ranked fronts in several steps (see lines 14–20). First, the algorithm verifies that there is enough space in the child population to add all the chromosomes in each ranked front $rank_k^f$ by comparing the remaining space in the child population $(n - length(Pop^c))$ with the rank front size $length(rank_k^f)$ (see line 16). If there is enough space, the rank front is directly added to the child population (see line 17). If there is no space, then the algorithm identifies the most prominent chromosomes in the front using a crowd comparison sort [28] (see line 19, function *CCS*) and assigns them to the child population. Through the loop of lines 15–20, the algorithm filters out the chromosomes in the total population Pop^t with the highest objective fitness values. The new population is used as the next parent population and again synthesizes a new child population by performing crossover and mutation (see lines 21–22). Finally, the non-dominated front (the Pareto optimal solution) $Rank^f$ is returned to the user which constitutes the clustering of BOs and operation nodes in the system to develop MSs (see line 24).

5 Implementation and Validation

In order to validate our microservice discovery and optimization process, a recommender[5] was developed based on the algorithms presented in Section 4 and we experimented with it on the SugarCRM and ChurchCRM customer relationship management systems. A detailed description of the experiments conducted on both systems is presented in this section.

SugarCRM as a system contains 8116 source files and 600 attributes divided between 101 tables, while ChurchCRM contains 8039 source files and 350 attributes divided between 55 tables. We generated execution sequences for both systems covering major functionalities[6] such as campaign management, customer management, etc. The execution logs containing the details about execution sequences, operations and database tables were captured using the log generation functionality already available in the systems. These execution logs cannot be directly used by process mining tools to obtain call graphs. Instead we used our own code[7] to convert them into XES format which is accepted by the Disco process mining tool. These XES files were then analyzed using Disco[8] and call graphs were generated for SugarCRM with 178 unique nodes and for ChurchCRM with 58 unique nodes. Each node in a call graph represents a unique operation performed on database tables in the system and the edges between nodes represent the number of calls between the nodes, similar to Fig. 4.

Discovered MSs: As the initial step, the prototype identified 18 different business objects related to SugarCRM, such as ‘action control lists’, ‘calls’, ‘contacts’, ‘campaigns’, ‘meetings’, ‘users’, ‘prospects’, ‘accounts’, ‘documents’, ‘leads’, ‘emails’, ‘projects’ and ‘email management’, 11 different business objects related to ChurchCRM, such as ‘calendar’, ‘locations’, ‘deposits’, ‘emails’, ‘events’, ‘family’, ‘group’, ‘property’, ‘query’, ‘users’ and ‘kiosk’. The identified BOs and the call graphs with execution details were given to the optimization algorithm described in Section 4 and tested against both scenarios depicted in Fig. 2 and 3. Both executions provided the same solution deriving 8 MSs for ChurchCRM and 11 MSs for SugarCRM.

Validation Process: Validation of the microservice recommendations was conducted in two steps. First, we evaluated the improvement of cohesion and coupling of different modules when clustering the classes based on recommendations provided by our prototype. This was achieved through measuring the Lack of Cohesion (LOC) and Structural Coupling (StrC) of the clusters, as described by Candela *et al.* [5]. We calculated the values for the enterprise system by clustering the classes into folders while preserving the original package structure, and then calculated the values for microservices. The

[5] https://github.com/AnuruddhaDeAlwis/NSGAIIFOROptimization.git
[6] http://support.sugarcrm.com/Documentation/Sugar_Versions/8.0/Pro/Application_Guide/
[7] https://github.com/AnuruddhaDeAlwis/XESConvertor.git
[8] https://fluxicon.com/disco/

LOC and StrC values calculated for ChurchCRM are summarized in Table 1 and Table 2, respectively, and LOC and StrC values calculated for SugarCRM are summarized in Table 3 and Table 4, respectively.

Next we validated the performance of the systems by implementing the microservices suggested by the prototype. In order to achieve this, first we hosted the SugarCRM and ChurchCRM systems in AWS Cloud. For each system, we used 2 EC2 instances which individually contained one virtual CPU and memory of 1GB. The data related to the systems were hosted in a MySQL relational database in AWS, which has one virtual CPU and storage of 20GB. A clear idea of this implementation can be obtained through the 'Enterprise System' section depicted in Fig. 5. These systems were then tested against 200 and 400 executions generated by 4 machines simultaneously for ChurchCRM and 100 and 200 execution generated by 4 machines simultaneously for SugarCRM, simulating customer requests, while recording their total execution time, average CPU consumption, and average network bandwidth consumption. For SugarCRM, we simulated the functionality related to *Campaign management*, while for ChurchCRM we simulated the functionality related to *People management*. For the simulations, we used Selenuim[9] scripts which executed the system similar to a real user. The Average CPU consumption of EC2 instances and DB instances and Average Network usage for ChurchCRM and SugarCRM enterprise system are listed in the first two rows of Table 5 and 9, respectively.

After obtaining the results for the enterprise systems, we needed to evaluate the effectiveness of introducing a single microservice (i.e., the scenario depicted in Fig.2) to the system based on the suggestions provided the by the prototype. As such, for SugarCRM we introduced a microservice which manages 'prospect' BO with its subset of operations suggested by the prototype and for the ChurchCRM we introduced a microservice which manages 'family' BO with its subset of operations suggested by the prototype. Each microservice was hosted on an AWS elastic container service (ECS), which has two virtual CPUs and a total memory of 2GB, as depicted on the right side of Fig. 5. The data related to the BOs of each microservice was stored in one MySQL relational database instance with one virtual CPU and total storage of 20GB. Next, the executions were performed on both enterprise systems again. Since microservices are extended parts of the enterprise systems in these executions, the enterprise systems used API calls to pass the data to the microservices and the microservices processed and sent back the data to the enterprise systems. The data in the microservice databases and enterprise system databases were synchronized using the Amazon database migration service replication instance. Then, we again conducted the same set of experiments while introducing operations to the 'prospect' BO microservice and 'family' BO microservice which contradicted the optimal suggestions given by the prototype. In this situation we introduced operations related to 'user' BO to both microservices. The objective of this is to validate the effectiveness of the suggestions provided by the prototype (i.e., to evaluate the effectiveness of clustering operations with BOs as suggested by the prototype). The results obtained for ChurchCRM and SugarCRM regarding this experiment are summarised in Tables 5 and 9, respectively. In the tables 'Legacy &

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

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

System Type	1	2	3	4	5	6	7	8	9	10	11
Original ES	61	188	853	7	4	1065	31	378	3064	13	17
MSs	61	77	666	33	8	1453	73	351	3802	3	10

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

System Type	1	2	3	4	5	6	7	8	9	10	11
Original ES	41	26	61	17	16	70	29	31	123	27	19
MSs	41	25	8	37	20	64	33	31	121	3	7

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

System Type	1	2	3	4	5	6	7	8	9	10	11	12
Original ES	42	342	229	65	63	581	53	26	64	14	64	33
MSs	11	291	208	65	42	547	53	26	64	14	53	33

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

System Type	1	2	3	4	5	6	7	8	9	10	11	12
Original ES	22	58	32	21	30	57	20	17	31	20	19	48
MSs	12	57	32	21	29	57	20	17	31	20	19	48

Table 5. Legacy vs single MS results for ChurchCRM.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
Original ES	200	29768	2580	4.27	1.6	6.57
Original ES	400	37579	4440	5.14	1.68	4.06
ES & Single MS (1)	200	28490	2220	3.06	2.365	11.42
ES & Single MS (1)	400	39620	4200	2.945	2.26	9.26
ES & Single MS (2)	200	33462	2340	3.153	2.25	11.45
ES & Single MS (2)	400	36936	4380	3.04	2.125	11.322

Table 6. Legacy vs Single MS System characteristics comparison for ChurchCRM.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [200]	Availability [400]	Efficiency [200]	Efficiency [400]
Original ES	2.819	2.459	1.448	97.727	97.368	1.000	1.000
ES & Single MS (1)	2.477	2.459	2.087	97.37	97.228	1.162	1.057
ES & Single MS (2)	3.061	2.998	3.138	97.5	97.33	1.103	1.014

Table 7. Legacy vs Multiple MS results for ChurchCRM.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
Original ES	200	29768	2580	4.27	1.6	6.57
Original ES	400	37579	4440	5.14	1.68	4.06
ES & Multi MS (1)	200	25900	2100	2.397	2.567	7.46
ES & Multi MS (1)	400	35713	4260	2.737	2.29	8.521
ES & Multi MS (2)	200	26040	2100	2.313	2.14	7.72
ES & Multi MS (2)	400	35926	4260	2.325	2.15	7.118

Table 8. Legacy vs Multiple MS System characteristics comparison for ChurchCRM.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Avail-ability [200]	Avail-ability [400]	Effi-ciency [200]	Effi-ciency [400]
Original ES	2.819	2.459	1.448	97.727	97.368	1.000	1.000
ES & Multi MS (1)	3.407	2.662	3.408	97.222	97.26	1.229	1.042
ES & Multi MS (2)	2.997	2.992	2.751	97.222	97.26	1.229	1.042

Table 9. Legacy vs single MS results for SugarCRM.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
Original ES	100	16417	2400	4.27	1.6	6.57
Original ES	200	20632	3900	5.14	1.68	4.06
ES & Single MS (1)	100	14318	2100	4.738	1.67	6.496
ES & Single MS (1)	200	19847	3540	2.42	1.58	5.717
ES & Single MS (2)	100	16099	2160	6.0175	1.64	6.465
ES & Single MS (2)	200	22426	3960	5.26	2.1	6.341

Table 10. Legacy vs Single MS System characteristics comparison for SugarCRM.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Avail-ability [200]	Avail-ability [400]	Effi-ciency [200]	Effi-ciency [400]
Original ES	2.206	2.435	4.403	97.56	97.01	1.000	1.000
ES & Single MS (1)	1.0472	1.939	1.804	97.22	96.72	1.143	1.102
ES & Single MS (2)	2.109	3.089	2.366	97.29	97.05	1.111	0.984

Single MS (1)' stands for the implementation suggested by the prototype (i.e., the correct MS implementation) and the 'Legacy & Single MS (2)' stands for the implementation we did against the suggestion given by the prototype (i.e., the wrong MS implementation).

Next, we introduced two microservices to each system to experiment with the scenario depicted in Fig.3. For SugarCRM, we introduced two microservices in which one manages 'prospect' BO with its subset of operations and the other manages 'user' BO with its subset of operations. Similarly for ChurchCRM we introduced two microservices in

Table 11. Legacy vs Multiple MS results for SugarCRM.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
Original ES	100	16417	2400	4.27	1.6	6.57
Original ES	200	20632	3900	5.14	1.68	4.06
ES & Multi MS (1)	200	16417	1920	3.905	2.167	5.716
ES & Multi MS (1)	400	20632	3900	3.861	2.067	5.097
ES & Multi MS (2)	200	41856	2160	4.521	1.73	5.190
ES & Multi MS (2)	400	52920	4140	4.526	1.723	5.899

Table 12. Legacy vs Multiple MS System characteristics comparison for SugarCRM.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [200]	Availability [400]	Efficiency [200]	Efficiency [400]
Original ES	2.206	2.435	4.403	97.56	97.01	1.000	1.000
ES & Multi MS (1)	3.246	3.131	2.927	96.96	97.01	1.25	1.000
ES & Multi MS (2)	2.926	2.911	3.322	97.29	97.18	1.11	0.942

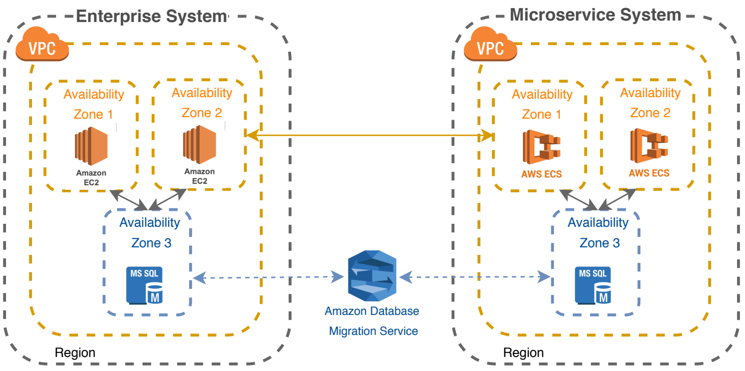


Fig. 5. System implementation in AWS for performance evaluation.

which one manages 'family' BO with its subset of operations and the other manages 'user' BO with its subset of operations. The configuration of the hardware and database properties of these microservices are similar to the ones we have set up for single microservices. Again, we executed the total system and obtained the results of 200 and 400 executions for ChurchCRM and 100 and 200 executions for SugarCRM. Then we again conducted the same set of experiments while introducing operations to 'prospect' BO, 'user' BO MSs of SugarCRM and 'family' BO,'user' BO MSs of ChurchCRM which contradict the optimal suggestions given by the prototype. As detailed earlier, the objective of this is to validate the effectiveness of the suggestions provided by the prototype. The results obtained for ChurchCRM and SugarCRM regarding this experiment are summarised in Tables 7 and 11, respectively. In the tables, 'Legacy & Multi MS (1)' stands for the implementation suggested by the prototype (i.e., the correct MS implementation) and the ‘Legacy & Multi MS (2)’ stands for the implementation we did against the suggestion given by the prototype (i.e., the wrong MS implementation).

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 6, 8, 10 and 12. Scalability was calculated according to the resources, usage over time, as described by Tsai *et al.* [30]. In order to determine availability, first we calculated the time taken to process 100 packets when a particular microservice is not available. Then, we measured the difference between the total uptime and total downtime, as described by Bauer *et al.* [31]. Efficiency gain was calculated by dividing the total time taken by the enterprise system to process all requests by the total time taken by the corresponding combined enterprise and microservice system.

Experimental Results: As described by Tsai *et al.* [30], the lower the number the better the scalability. Thus, it is evident from Tables 6, 8 10 and 12 that most of the time the microservices developed based on the suggestions provided by the prototype achieve better scalability than the ones we implemented contrary to the suggestions. When comparing availability, the gain is not significant. However, when comparing the execution efficiency of the systems it is clear from Tables 6, 8, 10 and 12 that the microservices developed based on the suggestions managed to process user requests quicker than the other systems, thus providing the output to the users more quickly.

The lower the lack of cohesion and structural coupling values the better the cohesion and coupling of the system [5]. As such, it is evident from Tables 1 and 4 that the microservices derived from the ChurchCRM and SugarCRM systems achieved better cohesion and coupling values than the legacy system. Thus, the obtained results have affirmed that the microservices extracted based on the suggestions provided by our prototype developed based on the algorithm in Section 4 led to microservices which could provide the same services to users while preserving overall system behaviour and achieving higher scalability, availability, efficiency, high cohesion, and low coupling.

6 Conclusion

This paper presented a novel technique based on queuing theory and business object relationships to support re-engineering of an enterprise system as microservices while improving system characteristics such as scalability, availability, cohesion and coupling. A prototype was developed based on the presented technique and validation was conducted by implementing the microservices recommended by the prototype for SugarCRM and ChurchCRM. The experiments conducted proved that the microservices derived based on the suggestions provided by the prototype had the desired characteristics. In future work, the presented technique can be further improved by evaluating method level relationships of the system.

References

1. Newman, S.: Building microservices: designing fine-grained systems. O’Reilly Media, Inc. (2015)
2. 2017 Internet Of Things (IoT) Intelligence Update, https://www.forbes.com/sites/louiscolumbus/2017/11/12/2017-internet-of-things-iot-intelligence-update/#43aa6f4c7f31. Last accessed 5 May 2018

3. Magal, S.R. and Word, J.: Integrated business processes with ERP systems. 1st edn. Wiley Publishing, (2011)
4. Anquetil, N. and Laval, J.: Legacy software restructuring: Analyzing a concrete case. In Software Maintenance and Reengineering (CSMR). In: Software Maintenance and Reengineering (CSMR) 15th European Conference, pp. 279–286 (2011)
5. Candela, I., Bavota, G., Russo, B. and Oliveto, R.: Using cohesion and coupling for software remodularization: Is it enough?. In: ACM Transactions on Software Engineering and Methodology (TOSEM), pp. 24. (2016)
6. Shatnawi, A., Seriai, A.D., Sahraoui, H. and Alshara, Z.: Reverse engineering reusable software components from object-oriented APIs. In: Journal of Systems and Software, pp. 442–460. (2017)
7. Balalaie, A., Heydarnoori, A. and Jamshidi, P.: Migrating to cloud-native architectures using MSs: an experience report. In: European Conference on Service-Oriented and Cloud Computing, pp. 201–215. Springer (2015)
8. Microservices a definition of this new architectural term, https://martinfowler.com/articles /microservices.html. Last accessed 3 May 2018
9. Evans, E.: Domain-driven design: tackling complexity in the heart of software, 1st edn. Addison-Wesley Professional (2003)
10. Nooijen, E.H.J, van Dongen, B. F. and Fahland, D.: Automatic discovery of data-centric and artifact-centric processes. In: International Conference on Business Process Management, pp. 316–327. Springer (2012)
11. Lu, X., Nagelkerke, M., van de Wiel, D. and Fahland, D.: Discovering interacting artifacts from ERP systems. In: IEEE Transactions on Services Computing, pp. 861–873. (2015)
12. Wei, F., Ouyang, C. and Barros, A.: Discovering behavioural interfaces for overloaded web services. In: Services (SERVICES), 2015 IEEE World Congress, pp. 286–293 (2015)
13. PrinciplesOfOod, http://www.butunclebob.com/ArticleS.UncleBob. PrinciplesOfOod. Last accessed 7 May 2018
14. De Alwis, A.A.C., Barros, A., Fidge, C. and Polyvyanyy, A., 2018, October. Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 60-79). Springer, Cham. (LNCS, volume 11230)
15. De Alwis, A.A.C., Barros, A., Polyvyanyy, A. and Fidge, C.: 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. Salah, K., Calyam, P. and Boutaba, R.: Analytical model for elastic scaling of cloud-based firewalls. In: Transactions on Network and Service Management, pp.136–146. IEEE (2017)
17. Klock, S., Van Der Werf, J.M.E., Guelen, J.P. and Jansen, S.: Workload-based clustering of coherent feature sets in microservice architectures. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 11–20. IEEE (2017, April)
18. Patidar, K., Gupta, R. and Chandel, G.S.: Coupling and cohesion measures in object oriented programming. In : International Journal of Advanced Research in Computer Science and Software Engineering, (2013).
19. Bauer, E. and Adams, R.: Reliability and availability of cloud computing. John Wiley & Sons (2012)
20. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M. and Stoica, I.: Quantifying eventual consistency with PBS. In: The VLDB Journal, pp.279–302. (2014)
21. Khazaei, H., Barna, C., Beigi-Mohammadi, N. and Litoiu, M.: Efficiency analysis of provisioning microservices. In: IEEE International Conference on Cloud Computing Technology and Science, pp. 261–268. IEEE (2016)

22. Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, M., Tantawi, A. and Youssef, A.:Performance management for cluster based web services. In: Integrated Network Management VIII, pp. 247–261. Springer (2003)
23. Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. and Steinder, M.: Performance evaluation of microservices architectures using containers. In: Network Computing and Applications (NCA), pp. 27–34. IEEE (2015)
24. Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: Performance Analysis of Systems and Software, pp. 171–172. IEEE (2015)
25. Huber, N., von Quast, M., Hauck, M. and Kounev, S.: Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In: CLOSER, pp. 563–573. (2011)
26. Lehrig, S., Eikerling, H. and Becker, S.: Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In : Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 83–92. ACM (2015)
27. Herbst, N.R., Kounev, S. and Reussner, R.H.: Elasticity in Cloud Computing: What It Is, and What It Is Not. In: ICAC, pp. 23–27. (2013)
28. Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T.A.M.T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. In: IEEE transactions on evolutionary computation, pp. 182–197. (2002)
29. Estrada, Z.J., Stephens, Z., Pham, C., Kalbarczyk, Z. and Iyer, R.K.: A performance evaluation of sequence alignment software in virtualized environments. In: Cluster, Cloud and Grid Computing (CCGrid), pp. 730–737. IEEE (2014)
30. Tsai, W.T., Huang, Y. and Shao, Q.: Testing the scalability of SaaS applications. In: Service-Oriented Computing and Applications (SOCA), IEEE International Conference, pp. 1–4. (2011)
31. Bauer, E. and Adams, R.: Reliability and availability of cloud computing, 1st edn. John Wiley & Sons (2012)