Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Genetic Algorithm to Find the Adequate Granularity for Service Interfaces

Daniele Romano and Martin Pinzger

Report TUD-SERG-2014-002

TUD-SERG-2014-002

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

A Genetic Algorithm to Find the Adequate Granularity for Service Interfaces

Daniele Romano
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
Email: daniele.romano@tudelft.nl

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
Email: martin.pinzger@aau.at

***Abstract*—The relevance of the service interfaces' granularity and its architectural impact have been widely investigated in literature. Existing studies show that the granularity of a service interface, in terms of exposed operations, should reflect their clients' usage. This idea has been formalized in the Consumer-Driven Contracts pattern (*CDC*). However, to the best of our knowledge, no studies propose techniques to assist providers in finding the right granularity and in easing the adoption of the *CDC* pattern.**

In this paper, we propose a genetic algorithm that mines the clients' usage and suggests Façade services whose granularity reflect the usage of each different type of clients. These services can be deployed on top of the original service and they become contracts for the different types of clients satisfying the *CDC* pattern. A first study shows that the genetic algorithm is capable of finding Façade services and it outperforms a random search approach.

***Keywords*-SOA; services; granularity; genetic algorithms;**

I. Introduction

One of the key factors for deploying successful services is assuring an adequate level of granularity [9], [4], [16], [8], [15]. The choice of how operations should be exposed through a service interface can have an impact on both performance and reusability [9], [16]. This level of granularity is also know in literature as *functionality granularity* [8]. For the sake of simplicity we refer to it simply as granularity throughout this paper. Choosing the right granularity is not a trivial task. On the one hand, fine-grained services lead their clients to invoke their interfaces multiple times worsening the performance [9], [4]. On the other hand, coarse-grained services can reduce reusability because their use is limited to very specific contexts [9], [4]. To find a trade-off between fine-grained and coarse-grained services the *Consumer-Driven Contracts* (*CDC*) pattern has been proposed [4]. This pattern states that the granularity of a service interface should reflect their clients' usage satisfying their requirements and becoming a contract between clients and providers.

In literature several studies have investigated the impact of granularity (*e.g.*, [9], [4], [16], [8], [15]), have classified the different levels of granularity (*e.g.*, [8]), and have proposed metrics to measure them (*e.g.*, [13], [3]). However, to the best of our knowledge, there are no studies proposing techniques to assist service providers in finding the right granularity and adopting the *CDC* pattern. This task can be expensive because many clients invoke a service interface in different ways. Providers should, first, analyze the usage of many clients and, then, design a service interface that satisfies all the clients' requirements.

In this paper, we propose a genetic algorithm to assist services providers in finding the adequate granularity and adopting the *CDC* pattern. This algorithm mines the clients' usage of a service interface and it retrieves Façade services [14] whose interfaces have an adequate granularity for each different type of clients. These Façade services become contracts that reflect clients' usage easing the adoption of the *CDC* pattern. Moreover, providers can deploy them on top of the existing service making this approach actionable without modifying it.

The contributions of this paper are as follows:

- a genetic algorithm designed to infer Façade services from clients' usage that represent contracts with the different types of clients.
- a study to evaluate the capability of the genetic algorithm compared to the capability of a random search approach.

The results show that the genetic algorithm is capable of finding Façade services and it outperforms the random search.

The remainder of this paper is organized as follows. Section II presents the problem and the proposed solution. Section III shows the proposed genetic algorithm. Section IV presents the study, its results, and discusses them. Related work is presented in Section V while in Section VI we draw our conclusions and outline directions for future work.

II. Problem Statement and Solution

In this section, first, we introduce the problem of finding the adequate granularity of services interfaces presenting the Consumer-Driven Contracts pattern. Then, we present our solution to address this problem.

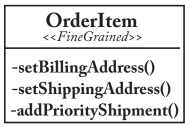
A. Problem Statement

Choosing the adequate granularity of a service is a relevant task and a widely discussed topic [9], [4], [16], [8],

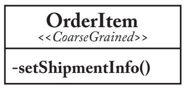
[15].

On the one hand, fine-grained services can lead to service-oriented systems with inadequate performance due to an excessive number of remote calls [9]. Consider for instance the fragment of a service interface to order an item shown in Figure 1. Figure 1a shows a fine-grained design for this service that exposes methods to set shipment and billing information for ordering an item. This design is efficient if the methods' invocation happens in a local environment (*e.g.*, in a software system deployed on a single machine) [9]. In a distributed environment (*e.g.*, in a service-oriented system) a client needs to invoke three methods (*i.e.*, *setBillingAddress()*, *setShippingAdress()*, and *addPriorityShipment()*) to set the needed information. This causes a significant communication overhead since three methods needs to be invoked over a network.

On the other hand, coarse-grained services interfaces can solve these issues. The coarse-grained *OrderItem* (shown in Figure 1b) exposes only one method (*i.e.*, *setShipmentInfo()*) to set all the information related to the shipment and the billing. In this way clients invoke the service only once reducing the communication overhead. However, if the services are too coarse-grained they can limit the reusability because their use will be limited to very specific contexts [9], [16], [4]. In our example in Figure 1, the clients of the coarse-grained service (Figure 1b) are constrained to set the billing address, the shipping address, and to add the priority shipment details. The service is not suitable for contexts where, for instance, priority shipments are not allowed. Maintenance tasks are needed to adapt coarse-grained services to different contexts. Hence, finding the adequate granularity of a service requires finding a trade-off between having a too fine-grained and having a too coarse-grained service. This allows to publish a service with an acceptable communication overhead and an adequate level of reusability.



(a) Fine-grained version exposing different methods for setting each different needed information.



(b) Coarse-grained version exposing a method to register all the needed information.

Figure 1: An example of fine-grained and coarse-grained service interfaces to set the shipping and the billing data for ordering an item.

To find such an adequate level of granularity the *Consumer-Driven Contracts (CDC)* has been defined for services interfaces [4]. The *Consumer-Driven Contracts (CDC)* pattern states that a service interface should reflect their clients needs through its granularity. In this way the service interface is considered a contract that satisfies the clients' requirements.

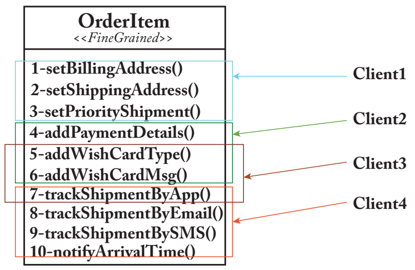


Figure 2: An example of a service interface to order an item for an e-commerce system. The rectangles represent independent methods that are invoked by a client.

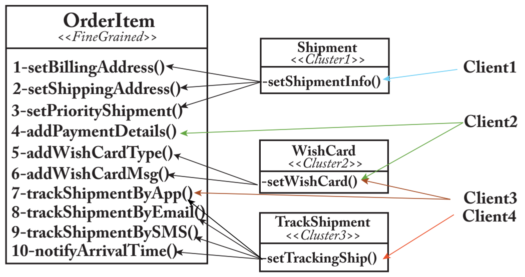
Applying the *CDC* pattern is not a trivial task. A service has usually several clients with different requirements invoking differently its interface. To deploy a service with an adequate granularity (using the *CDC* pattern) providers should know all these requirements. Within an enterprise or a corporate environment providers know their clients and they can understand how clients expect to use a service. However, clients are usually not known *a priori* and they bind a service only after it has been published and advertised. Moreover the number of clients and their different requirements can be huge and it can evolve over time.

B. Solution

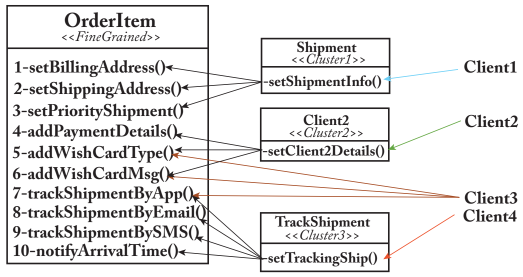
Our solution to the aforementioned problem consists in applying a cluster analysis. This analysis consists in clustering the set of methods in such a way that methods in the same cluster are invoked together by the clients. The goal of our cluster analysis is to find clusters that minimize the number of remote invocations to a service.

To better understand the cluster analysis for the granularity problem consider the example in Figure 2. The *OrderItem* extends the service shown in Figure 1a exposing further methods to 1) add payment details (*addPaymentDetails()*), 2) add a wish card to an order (*addWishCardType()* and *addWishCardMsg()*), and 3) to track the shipment (*trackShipmentByApp()*, *trackShipmentByEmail()*, *trackShipmentBySMS(), and* notifyArrivalTime()). Imagine this service has four clients (*Client1*, *Client2*, *Client3*, and *Client4*). These clients invoke different sets of independent methods denoted in Figure 2 by rectangles (*e.g.*, *Client1* invokes *setBillingAddress()*, *setShippingAddress()*, and *setPriorityShipment()*). These methods are considered independent because the invocation of one method does not require the invocation of the other ones [20]. In total there are 13 remote invocations: 3 performed by *Client1*, 3 by *Client2*, 3 by *Client3*, and 4 by *Client4*.

In this example we can retrieve three clusters (shown in Figure 3a) that minimize the number of remote invocations:



(a) The *Shipment*, *WishCard*, and *TrackShipment* have been introduced. This design has 9 local invocations and 6 remote invocations.



(b) The *Shipment*, *Client2*, and *TrackShipment* have been introduced. This design has 10 local invocations and 6 remote invocations.

Figure 3: Two possible refactorings of the service interface shown in Figure 2 using the proposed cluster analysis and using the Façade pattern. Black arrows indicate local invocations while non-black arrows indicate remote invocations.

- **Cluster1 (*i.e.*, Shipment):** consists of *setBillingAddress()*, *setShippingAddress()*, and *setPriorityShipments()*.
- **Cluster2 (*i.e.*, WishCard):** consists of *addWishCardType()* and *addWishCardMsg()*.
- **Cluster3 (*i.e.*, TrackShipment):** consists of *trackShipmentByApp() trackShipmentByEmail()*, *trackShipmentBySMS()* and *notifyArrivalTime()*.

Once we know the clusters we can combine the fine-grained methods belonging to a cluster into a single coarse-grained method. These coarse-grained methods can be exposed through Façade services [14] as shown in Figure 3a. Façade services (*i.e.*, *Shipment*, *WishCard*, and *TrackShipment* in our example) have been defined to provide different views of lower level services (*i.e.*, *OrderItem* in our example). Since the invocations from Façade services to lower-level services are local invocations (shown with black arrows in Figure 3), the total number of remote invocations (shown with non-black arrows in Figure 3) has been reduced to 6. Moreover, adopting this design choice allows us to keep public the fine-grained *OrderItem* whose methods can always be invoked by future clients and by current clients without breaking their behavior.

However, choosing the clusters that minimize the number of remote invocations cannot be performed fully-automatically. Imagine for instance that we change *Cluster2* adding the method *addPaymentDetails()* as shown in Figure 3b. This cluster is optimal for *Client2* that should perform only one remote invocation. However, *Client3* cannot invoke anymore the Façade service associated to the *Cluster2* because it contains a method (*i.e.*, *addPaymentDetails()*) in which it is not interested. The number of remote invocations is still equal to 6. At this point an engineer should decide which architectural design is more suitable for her specific domain. The decision might be influenced by three different factors:

- **Cohesion of Façade services:** the design in Figure 3a might be preferred because the *WishCard* service is more cohesive than the *Client2* service since it exposes related methods (methods related to the wish card concern).
- **Number of local invocations:** the design in Figure 3a might be preferred because it has 9 local invocations while the design in Figure 3b has 10 local invocations.
- **Relevance of different clients:** the service provider might want to give a better service (*e.g.*, upon a higher registration fee) to *Client2* and, hence, adopt the design in Figure 3b.

C. *Contributions*

In this paper we propose a search-based approach to retrieve the clusters of methods that minimize the number of remote invocations. As explained previously, the methods belonging to the same cluster can be exposed through a Façade service whose granularity reflects clients' usage and, hence, satisfy the *CDC* pattern.

A first approach to find these clusters consists in adopting brute-force search techniques. These techniques consist of enumerating all possible clusters and checking whether they minimize the number of invocations. The problem of these approaches is that the number of possible clusters can be prohibitively large causing a combinatorial explosion. Imagine for instance to adopt this approach for finding the right granularity of the *AmazonEC2* web service. This web service exposes 118 methods in the version 23 [18]. The number of 20-combinations of the 118 methods in *AmazonEC2* are equal to:

$$\binom{118}{20} = \frac{118!}{20!98!} \approx 2 * 10^{21}$$

This means that for only evaluating all the clusters with 20 methods the search will require executing at least $2 * 10^{21}$ computer instructions, which will take several days on a typical PC. Moreover, we should evaluate clusters with size ranging from 2 to 118 causing the number of computer instructions to further increase.

To solve this issue we propose a genetic algorithm (shown in Section III) that mimicking the process of natural selection finds optimal solutions (*i.e.*, cluster that minimize the number of remote invocations) in acceptable time without requiring special hardware configurations (*e.g.*, the use of supercomputers).

Moreover, we perform a first study aimed at investigating the capability of the proposed approach in finding Façade services that is presented in Section IV.

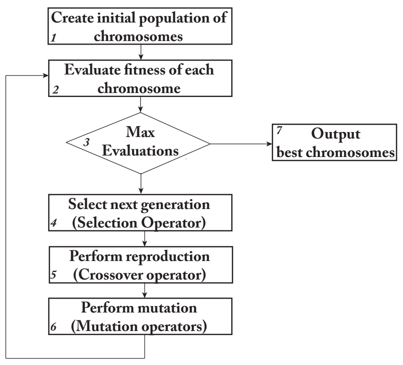
In this paper we do not cover the problem of mining independent methods because it has already been subject of related work [20] that can be integrated in our approach. Furthermore, related work [20] shows that 78.1% of the methods in their analyzed web services are independent. This percentage shows that most of the methods can be potentially joined in coarse-grained methods, motivating further the need of performing this task with a proper approach.

III. The Genetic Algorithm

Genetic Algorithms (GAs) have been proposed by Holland [10] and they have been used in a wide range of applications where optimization is required. Among all the applications GAs have been widely studied to solve clustering problems [11].

The key idea of GAs is to mimic the process of natural selection providing a search heuristic technique to find solutions to optimization problems. A generic GA is shown in Figure 4.

Differently to other heuristics (*e.g.*, *Random Search* and *Brute-Force Search*) that consider one solution at a time, a GA starts with a set of candidate solutions, also known as population (step 1 in Figure 4). These solutions are randomly generated and they are often referred to as chromosomes. Since the search is based upon many starting points, the likelihood to sample more of the search space is higher than local searches. This feature narrows down the likelihood to get stuck in a local optimum point in the search space. Each solution is evaluated through a fitness function that measures how ”good” a candidate solution is relatively to other candidate solutions (step 2). Solutions from the population are used to form new populations, also known as generations. This is achieved using the evolutionary operators. Specifically, first a pair of solutions (parents) is selected from the population through a selection operator (step 4). From these parents two offspring solutions are generated through the crossover operator (step 5). The crossover operator is responsible to generate offspring solutions that combine features from the two parents. To preserve the diversity, the mutation operators (step 6) mutate the offspring. These mutated solutions are added in the population replacing solutions with the worst fitness scores. This process of evolving the population is repeated until some condition (*e.g.*, reaching of max number of fitness evaluations in step 3 or achievement of the goal). Finally, the GA outputs the best solutions when the evolution process terminates (step 7).



1 Create initial population of chromosomes
2 Evaluate fitness of each chromosome
3 Max Evaluations
7 Output best chromosomes
4 Select next generation (Selection Operator)
5 Perform reproduction (Crossover operator)
6 Perform mutation (Mutation operators)

Figure 4: Different steps of a genetic algorithm.

To implement the GA and adapt it to find the set of clusters that minimize the number of remote invocations we have to define the fitness function, the chromosome (or solution) representation, and the evolutionary operators (*i.e.*, selection, crossover, and mutation).

A. Chromosome representation

To represent the chromosomes we use a *label-based integer encoding* widely adopted in literature [11] and shown in Figure 5. According to this encoding, a solution is an integer array of *n* positions, where *n* is the number of methods exposed in a service. Each position corresponds to a specific method (*e.g.*, position 1 corresponds to the method *setBillingAddress()* in Figure 2). The integer values in the array represent the cluster to which the methods belong. For instance in Figure 5, the methods 1,2, and 10 belong to the same cluster labeled with 1. Note that two chromosomes can be equivalent even though the clusters are labeled differently. For instance the clusters [1,1,1,1,2,2,2,2,3,3] and [2,2,2,2,3,3,3,3,1,1] represent the same clusters. To solve this problem we apply the *renumbering procedure* as shown in [5] that transforms different labelings of equivalent clusterings into a unique labeling.

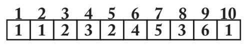


Figure 5: Chromosome representation of our candidate solutions.

B. Fitness

The fitness function is a function that measures how ”good” a solution is. Our fitness function counts for each chromosome the number of remote invocations needed by the clients. Imagine that the clients’ usage information of Figure 2 are saved in the data set shown in Table I.

In this data set, each row contains the id of the client (*i.e.*, *ClientID*) and the set of independent methods invoked by it (*i.e.*, *InvokedMethods*). The *InvokedMethods* are sets of methods where each integer value corresponds to a different method in the service. We labeled the methods in the *OrderItem* (shown in Figure 2) from 1 to 10 depending on

ClientID	InvokedMethods
Client1	1;2;3
Client2	4;5;6
Client3	5;6;7
Client4	7;8;9;10

Table I: Data set containing independent methods invoked by each different client in Figure 2.

the order they are declared in the service (*e.g.*, *setBillingAddress()* is labeled with *1*, *setShippingAddress()* is labeled with *2*, *etc.*).

Once we have this data set, we compute the fitness function as the sum of the number of remote invocations required to invoke each *InvokedMethods* set in the data set. If the methods (or a subset of methods) in an *InvokedMethods* set belong to a cluster containing no other methods, the methods in this cluster account for 1 invocation in total. Otherwise each different method accounts for 1. Consider for instance the chromosome [1,1,1,1,2,2,2,2,3,3]. This chromosome clusters together the methods 1, 2, 3, and 4 (*i.e.*, cluster 1), the methods 5, 6, 7, and 8 (*i.e.*, cluster 2), and the methods 9 and 10 (*i.e.*, cluster 3). In this case the number of remote invocations to execute the *InvokedMethods* of *Client1* (*i.e.*, 1;2;3) is 3 because the cluster 1 contains the method 4 that is not needed by it. Hence, *Client1* cannot invoke the Façade service represented by the cluster labeled *1* and it should invoke the methods in the original service *OrderItem*. If we change the chromosome into [1,1,1,2,2,2,2,2,3,3], the total number of invocations is equal to 1 because *Client1* can execute once the Façade service represented by the cluster *1*. If the chromosome becomes [1,1,2,2,2,2,2,2,3,3] then the total number of remote invocations is equal to *2*. The client invokes once the cluster *1* to invoke the methods *1* and *2*. Then it should invoke method *3* in the original service.

C. The Selection Operator

The selection operator selects two parents from a population according to their fitness. We use the Ranked Based Roulette Wheel (*RBRW*) that is a modified roulette wheel selection operator as proposed by Al Jadaan *et al.* [1]. *RBRW* ranks the chromosomes in the population by the fitness value: the highest rank is assigned to the chromosome with the best fitness value. Hence, the best chromosomes have the highest probabilities to be selected as parents.

D. The Crossover Operator

Once the GA has selected two parents (*ParentA* and *ParentB*) needed for generating the offspring, the crossover operator is applied to them with a probability P_c. As crossover operator we use the operator defined specifically for clustering problems by Hruschka *et al.* [11]. In order to illustrate how this operator works consider the example shown in Figure 6 from [11]. The operator first selects randomly k ($1 \leq k \leq n$) clusters from *ParentA*, where n is the number of clusters in *ParentA*. In our example assume that the clusters 2 and 3 are selected from *ParentA* (marked in red in Figure 6). The first child (*ChildC*) originally is created as copy of the second parent *ParentB* (step 1). As second step, the selected clusters (*i.e.*, 2 and 3) are copied into *ChildC*. Copying these clusters changes the clusters 1, 2, and 3 in *ChildC*. These changed clusters are removed from *ChildC* (step 3) leaving the corresponding methods unallocated (labeled with 0). In the forth step the unallocated methods are allocated to the cluster with the nearest centroid.

The same procedure is followed to generate the second child *ChildD*. However, instead of selecting randomly k clusters from *ParentB*, the changed clusters of *ChildC* (*i.e.*, 1,2, and 3) are copied into *ChildD* that is originally a copy of *ParentA*.

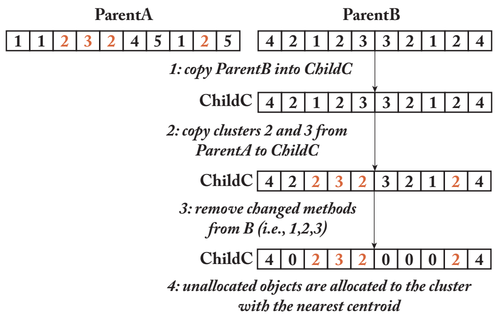

ParentA
ParentB
1 1 2 3 2 4 5 1 2 5
4 2 1 2 3 3 2 1 2 4
1: copy ParentB into ChildC
ChildC 4 2 1 2 3 3 2 1 2 4
2: copy clusters 2 and 3 from ParentA to ChildC
ChildC 4 2 2 3 2 3 2 1 2 4
3: remove changed methods from B (i.e., 1,2,3)
ChildC 4 0 2 3 2 0 0 0 2 4
4: unallocated objects are allocated to the cluster with the nearest centroid

Figure 6: Example of crossover operator for clustering problems [11].

E. The Mutation Operators

After obtaining the offspring population through the crossover operator, the offspring is mutated through the mutation operator with a probability P_m. This step is necessary to ensure genetic diversity from one generation to the next ones. The mutation is performed selecting one of the following cluster-oriented mutation operators (randomly selected) [5], [11]:

- **split**: a randomly selected cluster is split in two different clusters. The methods of the original cluster are randomly assigned to the generated clusters.
- **merge**: moves all methods of a randomly selected cluster to another randomly selected cluster.
- **move**: moves methods between clusters. Both methods and clusters are randomly selected.

F. Implementation

We implemented the proposed genetic algorithm on top of the *JMetal*[1] framework. *JMetal* is a Java framework that provides state-of-the-art algorithms for optimization problems. We calibrated the genetic algorithm as follows:

- the population is composed by 100 chromosomes. The initial population is randomly generated;
- the crossover and mutation probability is 0.9;

[1]http://jmetal.sourceforge.net

- the maximum number of fitness evaluation (step 3 in Figure 4) is 100,000.

These parameters were chosen using a trial-and-error procedure aimed at selecting values that, for our study shown in the next section, allowed to obtain the best results.

IV. Study

The *goal* of this study is to evaluate the capability of our approach in finding Façade services that minimize the number of remote invocations and reflect clients' usage. The *perspective* is that of services providers interested in applying the Consumer-Driven Contracts pattern using Façade services with adequate granularity. In this study we answer the following research question:

To which extent is the propose GA capable of identifying Façade services that minimize the number of remote invocations and reflect clients' usage?

In the following subsections, first, we present the analysis we performed to answer our research question. Then, we show the results and answer the research question. Finally, we discuss the results and the threats to validity of our study.

A. Analysis

To answer our research question we run the genetic algorithm (GA) defined in Section III to find the Façade services for the working example shown in Figure 2. To measure the performance of our GA we register the number of GA fitness evaluations needed to find the Façade services shown in Figure 3. Also, we compare the GA with a random search (RS), in which the solutions are randomly generated but no genetic evolution is applied. Both the GA and RS are executed 100 times and the number of fitness evaluations required to find the Façade services are compared through statistical tests. We use a random search as baseline because this comparison is considered the first step to evaluate a genetic algorithm [19]. Comparisons with other search-based approaches (*e.g.*, local search algorithms) will be subject of our future work.

First, we use the Mann-Whitney test to analyze whether there is a significant difference between the number of fitness evaluations required by the GA and the ones required by the RS. Significant differences are indicated by Mann-Whitney p-values $\geq$0.01. Then, we use the Cliff's Delta d effect size to measure the magnitude of the difference. Cliffs Delta estimates the probability that a value selected from one group is greater than a value selected from the other group. Cliffs Delta ranges between +1 if all selected values from one group are higher than the selected values in the other group and -1 if the reverse is true. 0 expresses two overlapping distributions. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$. We chose the Mann-Whitney test and Cliff's Delta effect size because they do not require assumptions about the variances and the types of the distributions (*i.e.*, they are non-parametric tests).

#Methods	GA	RS
10	100%	82%
11	100%	70%
12	100%	65%
13	100%	35%
14	100%	20%
15	100%	10%
16	100%	0%
118	100%	0%

Table II: Percentage of successful executions in which GA and RS find the Façade services shown in Figure 3.

Moreover, to analyze the capability of the GA in finding Façade services for bigger services, we increase stepwise the number of methods declared in the *OrderItem* in (shown in Figure 2). For each different size of the *OrderItem* we perform the same analysis: 1) we execute 100 times the GA and RS, 2) we register the number of fitness evaluations needed for finding the Façade services shown in Figure 3, and 3) we perform the Mann-Withney and Cliff's Delta test to analyze statistically the differences between the distributions. We increment the size of the service up to 118 methods, that is the size of the biggest WSDL interface (*AmazonEC2*) analyzed in our previous work [18].

B. Results

Table II shows the percentage of executions in which GA and RS find the right Façade services shown in Figure 3. The results show that, while the GA is always capable of finding the Façade services, the capability of the RS decreases with increasing numbers of methods. For services with 16 or more methods the RS is not capable to find the Façade services.

The number of fitness evaluations required by the GA and RS are shown in the form of box plots in Figure 7. The median number of fitness evaluations for the *OrderItem* with 118 methods required by the GA (not shown in Figure 7) is equal to 5754 (with a median execution time of 295 seconds[2]). Comparing it to the median number of fitness evaluations for the service with 10 methods (*i.e.*, 1049 fitness evaluations with a median execution time of 34.5 seconds) shows that GA scales well with increasing numbers of methods.

Moreover, the distributions of the numbers of fitness evaluations required by the GA and the RS is statistically different as shown by the Mann-Whitney p-values (<0.01) in Table III. The magnitude of these differences is always large as shown by Cliff's Deltas d (=1) in Table III. All the distributions, except RS12 in Figure 7, are not normally distributed (normality has been tested with the *Shapiro* test

[2]Execution times has been evaluated on a MacBook Pro Mid 2010, processor 2.66 GHz Intel Core i7, memory 4 GB 1067 MHz DDR3, OS 10.8.5.

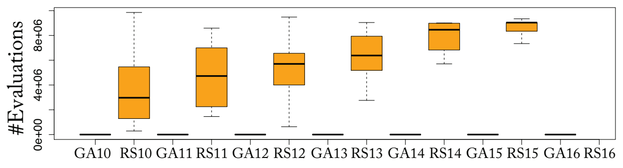


Figure 7: Box plots showing the number of fitness evaluations (#Evaluations) required by GA and RS. GAX and RSX label the box plots for the *OrderItem* with X methods.

#Methods	MW *p-value*	Cliff *d*
10	$<$ 2.2e-16	1
11	$<$ 2.2e-16	1
12	$<$ 2.2e-16	1
13	$<$ 2.2e-16	1
14	$<$ 2.2e-16	1
15	$<$ 2.2e-16	1

Table III: Mann-Whitney p-values (MW *p-value*) and Cliff's Delta *d* (Cliff *d*) between the distribution of #Evaluations required by the GA and RS.

and a confidence level of 0.05). As a consequence the non-parametric tests used in our analysis are the most suitable for these distributions.

Based on these results, we can answer our research question stating that the GA is capable to find Façade services outperforming the RS approach.

C. Discussions

The results of this study show that the proposed GA, differently to the RS, is capable to assist service providers in applying the Consumer-Driven Contracts pattern. Running the GA, providers can retrieve the Façade services that reflect the usage of their clients and minimize the number of remote invocations. Once the set of Façade services is retrieved, they should manually select the most appropriate Façade services as discussed in Section II. These Façade services can be deployed on top of the existing service without modifying it and preserving the compatibility of existing clients. Furthermore, since this approach is semi-automatic, it can be executed over time to monitor the evolution of clients' usage. This allows services providers to co-evolve the granularity of their services reflecting the evolving usage of their clients.

The main threats to validity that can affect our study are the threats to external validity. These threats concern the generalization of our findings. We evaluated our approach over a small working example. However, to best of our knowledge, there are no data sets that contain service usage information suitable for our analysis.

In literature different data sets are available for research on QoS (*e.g.*, [2], [21]). However, these data sets do not contain information about the operations invoked but only the services names and their url. As a consequence they are not suitable for our analysis.

V. Related Work

Granularity of services. The closest work to ours is the study developed by Jiang *et al.* [12]. In this study the authors propose an approach to infer the granularity of services mining the activities of business processes. The main idea consists of using frequent pattern mining algorithms to analyze the invocations to service interfaces. Our approach differs to theirs because it can mine the granularity of every kind of services and not only services involved in business processes. Furthermore we have not used the proposed frequent pattern mining algorithm because they require a special tuning of the *support* and *confidence* parameters that are problem specific. Moreover, these parameters, together with other relevant details, are not reported in [12] making the replication of this study not possible. To the best of our knowledge we are not aware of further studies aimed a inferring the right granularity of service interfaces.

Related work have mostly proposed classifications for different levels of granularity and have investigated metrics for measuring the granularity. Haesen *et al.* [8] have proposed a classification of three service granularity types (*i.e.*, *functionality*, *data*, and *business value* granularity). For each of these types they have discussed the impact on a set of architectural attributes (*e.g.*, performance, reusability and flexibility). In our paper we adhered to their *functionality* granularity that has been referred to as granularity for the sake of simplicity. Haesen *et al.* confirm that the *functionality* granularity can have an impact on both performance and reusability as stated in [9], [16], [4] and already discussed in Section II. Many other studies have investigated metrics to measure the granularity (*e.g.*, [13], [3]). For instance, Khoshkbarforoushha *et al.* [13] measure the granularity appropriateness with a model that integrates four different metrics that measures 1) the business value of a service, 2) the service reusability, 3) the service context-independency, and 4) the service complexity. Alahmari *et al.* [3] proposed a set of metrics to measure the granularity based on internal structural attributes (*e.g.*, number of operations, number of messages, complexity of data types). However, these studies are limited to measure the granularity and do not provide suggestions on inferring the right granularity.

Refactoring through genetic algorithms. Over the last years genetic algorithms, and in general search based algorithms, have become popular to perform refactorings of software artifacts. For instance, Ghannem *et al.* [7] found appropriate refactoring suggestions using a set of refactoring examples. Their approach is based on an Interactive Genetic Algorithm which enables to interact with users and integrate their feedbacks into a classic GA. Ghaith *et al.* [6] presented an approach to automate improvements of software security based on search-based refactoring. O'Keeffe *et al.* [17] have constructed a software tool capable of refactoring object-oriented systems. This tool uses search-based techniques to conform the design of a system to a given design quality model. These studies confirm that genetic algorithms are a useful technique to solve refactoring problems and satisfying desired quality attributes.

VI. Conclusion & Future Work

In this paper we have proposed a genetic algorithm to mine the adequate granularity of a service interface. According to the Consumer-Driven Contracts pattern, the granularity of a service should reflect its clients' usage. To adopt this pattern our genetic algorithm suggests Façade services whose granularity reflect the clients' usage. These services can be deployed on top of existing services allowing an easy adaptation of the Consumer-Driven Contracts pattern that does not require any modifications to existing services.

Our approach is semi-automatic as discussed in Section II. The genetic algorithm outputs different sets of Façade services that should be reviewed by providers. In our future work, first, we plan to further improve this approach to minimize the effort required from the user. Specifically, we plan to add parameters that can guide the search algorithm towards more detailed goals: giving more relevance to certain clients, satisfying other quality attributes (*e.g.*, high cohesion of Façade services, low number of local invocations), *etc.* Then, we plan to compare our genetic algorithm with other search-based techniques (*e.g.*, local search algorithms). Finally, we plan to improve the genetic algorithm suggesting overlapping Façade services that allow a method to belong to different Façade services. However, an *ad-hoc* study is needed to investigate to which extent the methods can be exposed through different Façade services because it can be problematic for the maintenance of service-oriented systems.

References

[1] C. R. O. Al Jadaan and L. Rajamani. Improved selection operator for ga. *Journal of Theoretical and Applied Information Technology*, 4(4), 2008.

[2] E. Al-Masri and Q. H. Mahmoud. Investigating web services on the world wide web. WWW, pages 795–804, New York, NY, USA, 2008. ACM.

[3] S. Alahmari, E. Zaluska, and D. C. De Roure. A metrics framework for evaluating soa service granularity. SCC, pages 512–519, Washington, DC, USA, 2011. IEEE Computer Society.

[4] R. Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Pearson Education, 2011.

[5] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[6] S. Ghaith and M. Ó. Cinnéide. Improving software security using search-based refactoring. In *SSBSE*, pages 121–135, 2012.

[7] A. Ghannem, G. El-Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *SSBSE*, pages 96–110, 2013.

[8] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans. On the definition of service granularity and its architectural impact. CAiSE, pages 375–389, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[11] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. C. P. L. F. De Carvalho. A survey of evolutionary algorithms for clustering. *Trans. Sys. Man Cyber Part C*, 39(2):133–155, Mar. 2009.

[12] J. Jiang, Y. Wu, and G. Yang. Making service granularity right: An assistant approach based on business process analysis. CHINAGRID, pages 204–210, Washington, DC, USA, 2011. IEEE Computer Society.

[13] A. Khoshkbarforoushha, R. Tabein, P. Jamshidi, and F. S. Aliee. Towards a metrics suite for measuring composite service granularity level appropriateness. In *SERVICES*, pages 245–252. IEEE Computer Society, 2010.

[14] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service Oriented Architecture Best Practices*. Prentice Hall, Upper Saddle River, NJ, 2005.

[15] N. N. Kulkarni and V. Dwivedi. The role of service granularity in a successful soa realization - a case study. In *SERVICES I*, pages 423–430. IEEE Computer Society, 2008.

[16] S. Murer, B. Bonati, and F. Furrer. *Managed Evolution - A Strategy for Very Large Information Systems*. Springer, 2010.

[17] M. O'Keeffe and M. í Cinnéide. Search-based refactoring for software maintenance. *J. Syst. Softw.*, 81(4):502–516, Apr. 2008.

[18] D. Romano and M. Pinzger. Analyzing the evolution of web services using fine-grained changes. In *ICWS*, pages 392–399, 2012.

[19] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[20] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Inferring dependency constraints on parameters for web services. In *WWW*, pages 1421–1432, 2013.

[21] Y. Zhang, Z. Zheng, and M. R. Lyu. Wsexpress: A qos-aware search engine for web services. In *ICWS*, pages 91–98. IEEE Computer Society, 2010.

TUD-SERG-2014-002
ISSN 1872-5392