Online Usage-aware Service Identification for Architectural Object-Oriented System Migration to SoA

Dionysis Athanasopoulos

School of Engineering & Computer Science
Victoria University of Wellington, New Zealand
dionysis.athanasopoulos@ecs.vuw.ac.nz

Abstract. Organisations that have relied their function on traditional systems have entered in the era of the digital connected world. The architecture of their systems is evolving to interconnected service-oriented systems that serve an abundance of clients. Since the plenitude of clients infers diverse and varying external usage of systems, the migration of systems to SoA should consider their external usage. It should further conform to fundamental design-principles that are related to the system usage, since their violation makes problematic the system evolution. However, existing service-identification approaches analyse only internal software-artifacts. Thus, we propose an approach that identifies services based on the external system-usage, conforming to the related design-principles. Since usage traces are not usually available beforehand, our approach is online, i.e. it iteratively (re-)identifies candidate services by the arrival of new usage-traces. The evaluation results of our approach on realistic case-studies show high effectiveness on identifying services that conform to the related design-principles.

Keywords: service identification, external usage, design principles, online process.

1 Introduction

Organisations that have relied their function on traditional systems have entered in the era of the digital connected world. The architecture of their systems is evolving to interconnected Service-oriented Architecture (SoA) systems [1] that serve an abundance of clients[1]. Unavoidably, this abundance of clients infers diversity in the external usage of systems. In particular, clients usually depend on different (subsets of) system functionalities. Moreover, system usage from the same clients may vary overtime. Thus, the migration of traditional systems to SoA should take into account the diverse and varying system-usage. While the architectural migration of Object-Oriented (OO) systems to SOA has been visited by the literature[2], *the common denominator of the existing approaches is that they identify services by analysing internal system artifacts (e.g. class dependencies), without considering the external system-usage that usually lead to the identification of different services than those based on internal artifacts*.

[1] We use the term client to refer to a piece of software that accesses a service.
[2] For a systematic literature review, the interested reader may refer to the recent survey in [2].

Considering the external system-usage, SoA system should consist of services that represent reusable functionalities with a single external reason to change (Single Responsibility Principle - SRP [3]). Moreover, (internal or external) service clients should not depend on (parts of) service interfaces that they do not use (Interface Segregation Principle - ISP [3]). In general, SRP and ISP violation makes problematic the evolution of systems of diverse and varying external usage. [4]. To the best of our knowledge, *there is no usage-aware service-identification approach for the architectural migration to SoA, let alone, an approach based on fundamental design-principles*.

Ideally, the usage-aware service-identification should define services specific for each group of clients. However, in our case, *such groups are not available beforehand and may vary overtime*. On top of this, *clients of the same group do not necessarily use exactly the same set of (front- and back-end) system methods*[3]. Thus, services should be re-identified in an online way by the dynamic arrival of new system usage-traces[4]. The naïve online approach that combines (groups of) clients that have been dynamically formed to system methods faces a (time and space) efficiency problem due to the plenitude of clients and the intractable (a.k.a. exponential) number of all the followed sequences of (front- and back-end) methods.

Contribution. We propose an *initial version of an efficient online usage-aware service-identification approach*. It keeps aggregated histories of *complete* method sequences[5] (i.e. usage traces), instead of keeping all the followed (sub-)sequences, and identifies services from usage-traces that are related to each other. Our intuition is that *related usage-traces have been followed by the same or related clients*, i.e. they use the same or subsets of system functionalities that depend on common methods. Overall, the approach does not form client groups, but it identifies services whose operations have been used at a high percentage by related clients.

Since the approach is online, it iteratively (re-)identifies services by the arrival of new usage-traces. To apply SRP, it identifies specific *method-invocation patterns* that indicate the existence of externally reusable services. To this end, the approach defines system-usage functions and checks the *monotonicity* of the latter. Finally, the approach applies ISP via merging services based on *related usage-traces* (followed by related clients). To do so, it uses the metric that we propose for assessing service usage-relatedness. Finally, the approach releases stable versions of services when the same set of services is identified for a fixed number of successively arrived usage-traces. Finally, we evaluate the effectiveness of the approach on realistic case-studies. The evaluation results show high effectiveness on producing services that conform to SRP and ISP.

Our contribution is summarized and structured as follows. Section 2 categorizes and compares related approaches. Section 3 describes our motivation example. Section 4 defines the representation models of OO and SoA systems, the system-usage artifacts, and the service usage-relatedness metric. Section 5 provides an overview of the initial version of our process and Section 6 describes the modus operandi of the process mechanisms. Section 7 evaluates the effectiveness of our approach on case-studies. Finally, Section 8 summarizes our approach and discusses its future research directions.

[3] We consider that front (resp. back)-end methods are (resp. not) directed invoked by clients.
[4] Off-line processes require all the data before the start of their execution.
[5] It starts with a front-end method and ends with a method that cannot invoke any other method.

Table 1. Summary and comparison of related service-identification approaches.

	Software Artifacts	Identification Process	Quality Metrics	Principles	Automation
[7–11]	internal system artifacts	off-line	X	X	guidelines
[12–14]					manual
[5, 15, 16]					(semi-)automated
[17]			cohesion & coupling		automated
Ours	*external usage*	*online*	usage cohesion	SRP & ISP	

2 Related Work

The migration to SoA typically includes the following sequence of phases [2]: system understanding, migration feasibility, service identification, technical evolution, and service deployment. Service identification can be realized from an architecture (e.g. [5]) and/or business perspective (e.g. [6]). In the second perspective, identified services should conform to all the business rules contained in OO systems. In the first perspective, the system architecture is re-designed. The realization of the re-design at a code level (e.g. method encapsulation) is performed in the technical-evolution phase.

We focus on categorizing and comparing the most representative approaches that deal with service identification from an architecture perspective[2]. While service identification is the most important phase in migration to SoA, a few automated approaches have been proposed in the literature. In detail, the majority of the approaches offers guidelines to identify candidate services (e.g. [7–11]) or proposes manual processes (e.g. [12–14]). The (semi-) automated approaches (e.g. [5, 15, 16]) adopt techniques, such as feature extraction [15], dominance analysis in directed graphs (e.g. [5]), and clustering methods [16]. Independently of the automation degree, all the approaches identify services by analysing *internal system artifacts*, such as requirements specification, system architecture, and source code. Moreover, all the approaches are *off-line*, since they require the used artifacts before the start of the service-identification process, without being able to improve the quality of identified services.

On top of this, the existing approaches, except for the recent one in [17], rely on ad-hoc criteria for evaluating service quality, inferring a potential gap between identified and expected services. [17] identifies candidate services based on metrics that assess the *cohesion* and *coupling of service implementation*. On the contrary, our service usage-relatedness metric is used for identifying services compliant to ISP via assessing the *cohesion of service usage*. In particular, our metric is related to the interface usage-cohesion metric in [18]. However, *the latter is applied only on front-end methods, assesses usage cohesion in terms of specific client groups whose number is low, and impractically considers that systems with singleton services (i.e. of single operation) have maximum cohesion* (see Section 7). Concerning the *external service reusability* (required by SRP), to the best of our knowledge, no such metric exists in the literature.

Comparing the related approaches in terms of the adopted system artifacts, process, quality metrics, principles, and automation degree (see Table 1), we observe that our approach is the only automated approach that (i) *considers external system-usage*, (ii) *evaluates system-usage cohesion*, (ii) *is online*, and (iii) *conforms to SRP and ISP*.

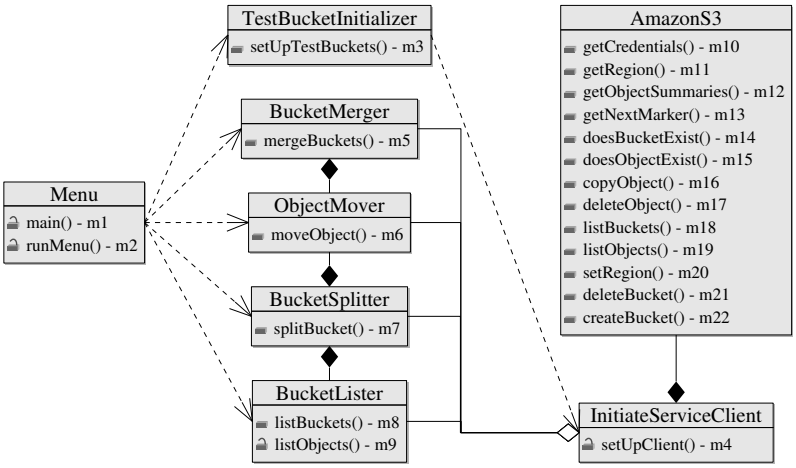


Fig. 1. The UML class diagram of the object-oriented implementation of the running example.

3 Motivation Example

We use an OO system that manages the organization of objects (e.g. text files, photos) into buckets and stores them on the Amazon cloud using the SDK of the `S3` service[6]. The system offers six functionalities, modelled as different classes and designed to be used by a high number of clients (`S3` offers scalable object storage). Fig. 1 depicts the UML diagram of the system implementation. `TestBucketInitializer` sets up test buckets and uses `InitiateServiceClient` to instantiate an `S3` client. `BucketMerger` merges the content of buckets using `ObjectMover` to move bucket objects. `BucketSplitter` splits a bucket into multiple ones using `BucketLister` for selecting which objects of the source bucket will be moved to each new bucket (reusing `ObjectMover`). `BucketLister` prints out the content of buckets. All the six classes depend on `AmazonS3` that is a part of the `S3` interface exposed by Amazon.

To migrate the system to SoA, we initially applied the approach in [17] that identifies services of high cohesion and low coupling based on class dependencies. Since coupling and cohesion among the classes `InitiateServiceClient`, `BucketMerger`, `ObjectMover`, `BucketSplitter`, and `BucketLister` is high, while the coupling between the last four classes and `TestBucketInitializer` is zero, the identified services are depicted in Fig. 2 (b). We also monitored the system usage and we observed that three groups of related end-users existed in our scenario, those that were (i) reorganizing their buckets using one of or both `BucketSplitter` and `BucketMerger`, (ii) keeping constant the scheme of their buckets and were moving objects using both `ObjectMover` and `BucketLister`, or (iii) producing test buckets, without storing their own objects, using `setUpTestBuckets`. Based on the end-user groups, we further expect that three groups of related clients exist. The expected set of services that conform to SRP and ISP are depicted in Fig. 2 (a), in which clients do not depend on

[6] aws.amazon.com/s3

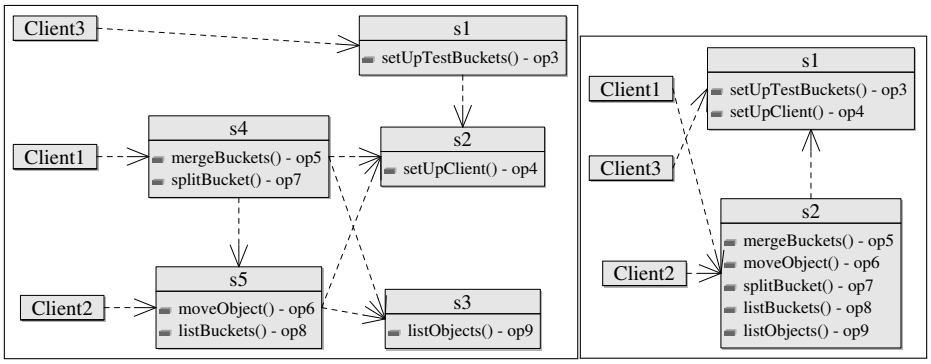


Fig. 2. Identified services for the system of the running example.

(front- and back-end) services that they do not use. The dependencies between the services are assigned based on the external method-usage. Concerning the services identified by [17] (Fig. 2 (b)), the clients depend on methods that they do not invoke, violating ISP. Concluding, approaches that consider internal system artifacts (e.g. class dependencies) identify services that do not necessarily reflect the external system-usage.

4 Basic Notions

We define the representation models of OO and SoA systems (Section 4.1), the system-usage artifacts (Section 4.2), and the service usage-relatedness metric (Section 4.3).

4.1 Representation Models of OO and SoA Systems

OO system representation. Independently of OO programming languages, our model represents an OO system as a directed rooted tree of methods (see Section 5). A method `m` (Table 2 (1)) is characterized by its `id`[7], `name`, and `isAccessorMutator` (getter/setter) fields. We keep the latter, since the usage of common accessors/mutators indicates that methods operate on the same data-models. The method usage is characterized by its total invocation number and the set of its directly invoked methods (fields `inv` and `edges` in Table 2 (1)). Our model does not depend on the system classes and packages. Fig. 3 (a) depicts the system representation of our running example.

SoA system representation. A service consists of its interface and implementation (fields `si` and `simpl` in Table 2 (2)). The service interface `si` (Table 2 (3)) is defined as a set of operations. Each operation `op` (Table 2 (4)) encapsulates an OO method and has the same `name`, `id`, and `inv` with the method. The service implementation `simpl` (Table 2 (5)) includes for each service operation the encapsulated OO method, along with the set of all the methods that are (directly or indirectly) invoked by the encapsulated method (stored in the set $internalM$ of Table 2 (5)), forming a subtree of

[7] Our system processor generates unique method identifiers to avoid name conflicts.

Table 2. The definition of the representation models of OO and SoA systems.

[**Method**]: $M := \Big(id : \texttt{int},\ name : \texttt{String},\ isAccessorMutator : \texttt{boolean},\ inv : \texttt{int}, edges\Big) \mid$ $edges = \Big(w_i \in [0,1],\ m_i : M\Big)$ (1)
[**Service**]: $S := \Big(si : SI,\ simpl : SIMPL\Big)$ (2)
[**Interface**]: $SI := \Big\{op_i : OP\Big\}$ (3)
[**Operation**]: $OP := \Big(id : \texttt{int},\ name : \texttt{String},\ inv : \texttt{int}\Big)$ (4)
[**Implementation**]: $SIMPL := \Big\{\Big(op_i : OP, internalM,\ externalOP\Big)\Big\} \mid$ $simpl.op_i.id = si.op_i.id \wedge internalM = \{m_k : M\} \wedge externalOP = \{op_j : OP\}$ (5)

the system representation. This subtree ends when leaf methods or nested services are met. The service implementation further includes the invoked operations of the nested services (stored in the set $exteranlOP$ of Table 2 (5)).

4.2 Usage Artifacts

To identify candidate services, our process aggregates past usage-traces to form their histories, checks for the existence of method-invocations patterns in the usage-trace histories, and also checks for related histories. These notions are formally defined below.

Definition 1. *[**Usage trace**] A usage trace* $\texttt{T}$ *is a sequence of methods,* $(id_1, id_2, \ldots, id_N)$*, that corresponds to a complete path*[8] *of the tree system-representation.* □

Definition 2. *[**Usage-trace history**] The history of* $\texttt{T}$ *is defined by a discrete function*[9] $\texttt{h}\colon \texttt{id} \rightarrow \texttt{inv} \in \mathbb{N}$ *that maps each method* $\texttt{id}$ *of* $\texttt{T}$ *to the method* $\texttt{inv}$*.* □

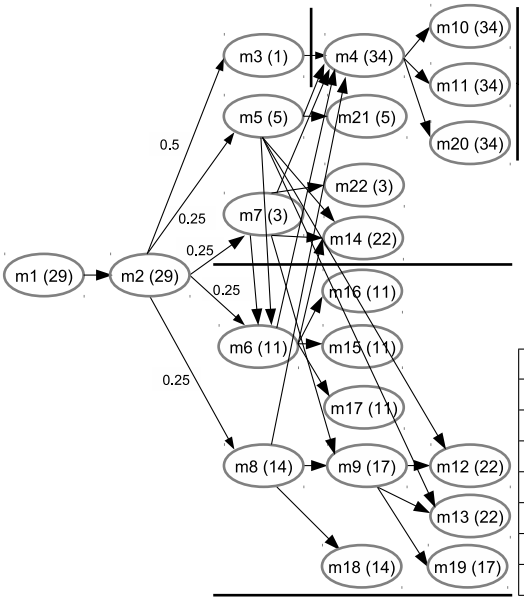
Fig. 3 (a) concisely depicts the histories of all usage-traces of the running example. Focusing on the usage traces, $(m_1, m_2, m_3, m_4, m_{10})$ and $(m_1, m_2, m_7, m_9, m_{12})$, Fig. 4 plots their histories. We observe from the figure that $\texttt{h}$ is not a monotonic function [19], but it is characterized by *local extrema* that help our mechanisms to identify *method-invocation patterns*, as explained in the following definitions. One such pattern appears when a method is used in all system executions. Thus, such a method belongs to system parts that do not define reusable services (e.g. client, controller code). Another pattern appears if a method is frequently reused. Then, it belongs to a service interface. About the service implementation, a pattern appears when two methods are used with close frequencies (higher than a threshold) by service operations. Another pattern appears when a frequently used method invokes methods of another service (nested services). These patterns are formally defined and exemplified below.

Definition 3. *[**Non-service pattern**] Let* id_x *be a method of a usage trace,* id_x *participates to a non-service pattern if* $\texttt{h}$ *at* id_x *equals to the system-execution number.* □

For instance, we observe in Fig. 4 that id_1 and id_2 do not belong to services (but to the client class $\texttt{Menu}$), since their $\texttt{h}$ values equal to 29 (total system-execution number).

[8] A path that starts from the root of the tree representation and ends to a tree leaf.

[9] Its domain values are distinct and unconnected (in contrast to continuous functions) [19].



(a) Usage-trace histories

	s1	s2	s3	s4	s5	s6	s7
s1	-	0.01	0.05	0.01	0.06	0.01	0.00
s2	0.01	-	0.02	0.17	0.02	0.11	0.00
s3	0.05	0.02	-	0.11	**0.39**	0.07	0.03
s4	0.01	0.17	0.11	-	0.09	**0.32**	0.00
s5	0.06	0.02	0.39	0.09	-	0.20	0.06
s6	0.01	0.11	0.07	0.32	0.20	-	0.26
s7	0.00	0.00	0.03	0.00	0.06	0.26	-

(b) Service usage-relatedness values

Fig. 3. The usage-trace histories and the service usage-relatedness values for the running example.

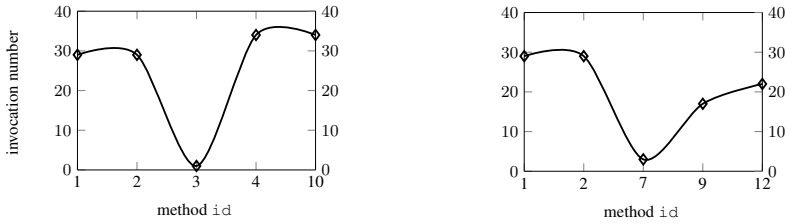
Definition 4. *[**Service-interface pattern**] Let id_x be a method of a usage trace, id_x belongs to a service-interface pattern if (i) it does not belong to a non-service pattern, (ii) is not accessor or mutator, (iii) $\mathtt{h}$ has local minimum at id_x or is decreasing at id_{x-1} and non-decreasing at id_{x+1} (if the latter exists).* □

Returning to Fig. 4 (a), we observe that id_3 belongs to a service interface, since $\mathtt{h}$ has local minimum at id_3. Also, id_4 belongs to another service-interface, since $\mathtt{h}$ is decreasing at id_3 and non-decreasing at id_{10}. Similarly, we observe in Fig. 4 (b) that id_7 belongs to a service interface, since $\mathtt{h}$ has local minimum at id_7. Moreover, id_9 belongs to another service-interface, since $\mathtt{h}$ is decreasing at id_7 and non-decreasing at id_{12}. However, id_{12} does not belong to a service interface, since it is an accessor.

Definition 5. *[**Service-implementation pattern**] Let id_x be a method of a usage trace, id_x belongs to a service-implementation pattern if (i) it does not belong to a non-service pattern, $\mathtt{h}$ is decreasing at id_{x-1} and non-decreasing at id_{x+1} (if the latter exists) or is constant at id_{x-1} and id_x.* □

Returning to Fig. 4 (a), we observe that id_4 (resp. id_{10}) belongs to a service implementation, since $\mathtt{h}$ has decreasing value at id_3 and non-decreasing at id_{10} (resp. constant value at id_4 and id_{10}). Similarly, we observe in Fig. 4 (b) that id_9 (resp. id_{12}) belongs to service implementation, since $\mathtt{h}$ has decreasing value at id_7 and non-decreasing at id_{12} (resp. decreasing value at id_9 - the value of $\mathtt{h}$ at id_{x+1} does not exist).

Overall, according to our patterns, a client software and four services, along with their implementations, were identified from Fig. 4 (a) and (b). We exploit the method-invocations patterns to identify reusable services that conform to SRP. We now turn to define how we can relate usage traces to identify services that conform to ISP.



(a) Usage trace $(m_1, m_2, m_3, m_4, m_{10})$ (b) Usage trace $(m_1, m_2, m_7, m_9, m_{12})$

Fig. 4. Plotting the histories of two usage traces of the running example.

Definition 6. ***[Related usage-trace histories]*** *Let h_1 and h_2 be the histories of T_1 and T_2, h_1 and h_2 are related if (i) $id_x = id_y$ or their values at $id_x \in T_1$ and $id_y \in T_2$ are close (higher than a threshold) and (ii) their usage-trace sub-sequences $(id_x, \ldots, T_1.length)$ and $(id_y, \ldots, T_2.length)$ include common methods.* □

Returning to Fig. 3 (a), we observe that the usage traces (m_1, m_2, m_5, m_{14}) and (m_1, m_2, m_7, m_{14}) have close values (compared to other method pairs) at m_5 and m_7 (inv = 5 vs. 3) and their sub-sequences (m_5, m_{14}) and (m_7, m_{14}) have a common method. The expected service s_4 of Fig. 2 (a), defined from these usage-traces, is exclusively used by `Client1`, conforming to ISP.

4.3 Metric of Service Usage-Relatedness

To calculate the usage-relatedness of two singleton services, we propose a metric that includes the parts R_1 and R_2 (Table 3). R_1 assesses the degree to which the methods that are encapsulated by the two service operations have been used a close number of times in usage traces (first part of Def. 6). To this end, R_1 calculates the percentage of the min divided by the max invocation numbers of the service operations. R_2 assesses the degree to which common methods of the service implementations were used in usage traces (second part of Def. 6). To this end, R_2 calculates the percentage of the number of the common methods (set interception), divided by the max implementation size, i.e. the highest number of methods. R_1 and R_2 are multiplied in R (Table 3), assuming that R has high values when both R_1 and R_2 have high values. Using R for merging services, the services have high usage cohesion (used by related clients), since their clients do not depend on services which they do not use.

5 Online Usage-aware Service-Identification Process

The proposed process includes the four mechanisms that are depicted in Fig. 5.

Processing OO System. The mechanism processes the source code of an OO system to represent it based on our model (Section 4.1). It examines the internal code of all the system methods[10] and keeps their directly invoked methods. In loop or condition blocks (e.g. `while`, `if`), it keeps the set of their directly invoked methods and further

[10] It excludes the programming-language or platform-dependent methods (e.g. `print()`).

Table 3. The definition of the metric of service usage-relatedness (on the absolute scale [0, 1]).

$$R(s_1,\ s_2) := R_1(s_1,\ s_2)\ *\ R_2(s_1,\ s_2) = \frac{MIN\Big(s_1.si.op.inv,\ \ s_2.si.op.inv\Big)}{MAX\Big(s_1.si.op.inv,\ \ s_2.si.op.inv\Big)}\ * \\ \frac{\Big|s_1.simpl.internalM\ \cap\ s_2.simpl.internalM\Big|}{MAX\Big(|s_1.simpl.internalM|,\ \ |s_2.simpl.internalM|\Big)}$$

annotates each method with a weight `w` (Table 2 (1)), which corresponds to the possibility of being invoked (Fig. 3 (a) depicts the weights that are smaller than 1.0). In loop blocks, the processor considers that the possibility is 0.5, while in condition blocks, the percentage of the condition branches. The processor breaks cycles via removing the invocations that close cycles. If a method is invoked multiple times by the same method, then the former is kept only once. The `main` method is the root of the tree model.

OO System Monitoring. The mechanism uses a typical monitoring agent (e.g. `jalen`[11]) to trace method invocations. The mechanism increases the total invocation number of the methods included in a usage trace. The mechanism is *space efficient*, since it does not store all the possible method (sub-)sequences. On the contrary, it annotates the system representation with the total invocation number of the methods (field `inv` in Table 2 (1)). Finally, the mechanism triggers the execution of the third mechanism by the arrival of a new usage trace, providing the annotated system-representation.

Vertical Service Identification. The mechanism checks for the existence of the method-invocation patterns (Def. 3-5) in usage-trace histories. Based on the patterns, the mechanism identifies services whose interfaces are singleton. The patterns are further used to identify the methods that implement the service interfaces. We hereafter call the mechanism *vertical* (see the two vertical lines in Fig. 3 (a)), since it identifies the front- (interface) and back-end methods (implementation) of services.

Horizontal Service Identification. The mechanism applies a clustering method to form groups of services with related usage-trace histories based on our service-relatedness metric. We hereafter call the mechanism *horizontal* (see the two horizontal lines in Fig. 3 (a)), since it identifies groups of services used by related clients.

6 Online Usage-aware Service-Identification Mechanisms

Vertical service-identification The mechanism defines services, complaint to SRP, via identifying method-invocations patterns in usage-trace histories. The underlying Algorithm 1 accepts the current total system-execution number `E` and the annotated tree system-representation, rooted at method m_0. The algorithm returns a list of singleton services `sList`. In detail, the algorithm efficiently traverses all the tree paths via adopting the classical Depth-First Search (`DFS`) algorithm (Alg. 1 (3-5)). While visiting a tree node, the algorithm constructs the path, to which the node belongs, that corresponds to a usage trace t (Alg. 1 (6-8)). For each method id_x of t (Alg. 1 (9)), the algorithm checks based on the history h of t, if id_x participates to the following patterns (Def. 3-5):

[11] github.com/adelnoureddine/jalen/tree/master/1.1

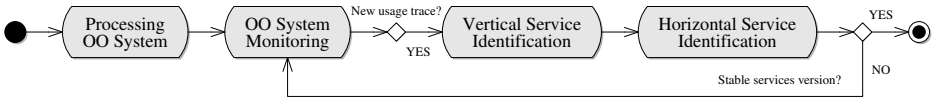


Fig. 5. The proposed online usage-aware service-identification process.

Non-service pattern: if the invocation number of the method id_x equals to `E`, then the method does not belong to a service interface or implementation (Alg. 1 (10)).

Service-interface pattern: if id_x is not accessor or mutator (Alg. 1 (11)), and if (i) h has local minimum at id_{x-1} (Alg. 1 (12)), i.e. the discrete derivative[12] of `h` at id_{x-1} is negative ($\Delta_{id_{x-1}} h < 0$) and positive at id_{x+1} ($\Delta_{id_{x+1}} h > 0$), or (ii) h is decreasing at id_{x-1} ($\Delta_{id_{x-1}} h > 0$) and non-decreasing at id_{x+1} ($\Delta_{id_{x+1}} h \geq 0$ – if it exists) (Alg. 1 (12)), then id_x belongs to a new service interface (Alg. 1 (14-16)).

Service-implementation pattern: if h is (i) decreasing at id_{x-1} ($\Delta_{id_{x-1}} h > 0$) and non-decreasing at id_{x+1} ($\Delta_{id_{x+1}} h \geq 0$ – if it exists)) (Alg. 1 (17)), or (ii) constant at id_{x-1} and id_x (Alg. 1 (17)), then id_x belongs to the last service (Alg. 1 (18-21)).

Complexity. The algorithm visits each method of the tree system-representation twice, one for constructing usage traces and one for checking the invocation patterns. Thus, the algorithm is *time efficient*, since its complexity scales with the linear expression, $\mathcal{O}(2 * |V| + |E|)$, which is the complexity of `DFS` (`|V|` is the methods number and `|E|` the tree edges number) and not with the intractable number of all tree paths.

Illustrative example. Returning to our running example, the algorithm examines the usage-trace histories depicted in Fig. 3 (a). Applying the previously described patterns, the algorithm identifies the singleton services, s_1–s_7, depicted in Table 4 (a).

Horizontal service-identification The mechanism merges services from related usage-traces (followed by related clients) so that services conform to ISP. The underlying Algorithm 2 accepts a set of singleton services `sList` and a `threshold` (of the lowest acceptable usage-relatedness value). The algorithm returns a list of (non-)singleton `services`. In detail, the algorithm calculates the usage-relatedness value of every pair of singleton services by using our metric R and stores them in the matrix r (Alg. 2 (1-2)). Since services can be merged in alternative ways, the algorithm adapts the classical agglomerative clustering-method [20]. It initially considers that each service forms a singleton cluster (Alg. 2 (3)) and repeatedly merges cluster-pairs until no more clusters can be merged or only one cluster remains (Alg. 2 (4-11)). To calculate the usage relatedness between two clusters (called proximity) [20], the algorithm adopts the min usage-relatedness metric (Alg. 2 (5-6)), since it guarantees that every service in a cluster is related to any other service. After merging two services (Alg. 2 (7)), the list `services` and the matrix r are accordingly updated (Alg. 2 (8-10)).

Complexity. The algorithm is *time efficient*, since its complexity equals to that of the classical agglomerative method, $\mathcal{O}(|V| * (|V| - 1)/2)$, where `|V|` is the number of singleton clusters. In the worst case, `|V|` equals to the number of the system methods.

Illustrative example. Given the singleton services of Table 4 (a), the algorithm identifies the services s_1–s_5 of Table 4 (b), which are the same with the expected ser-

[12] The discrete derivative of a function f at input n equals to $\Delta_n f = f(n+1) - f(n)$ [19].

Algorithm 1 Vertical service-identification

Input: `int` E, m_0
Output: `List<`S`>` $sList$

1: $T\ t \leftarrow \emptyset$
2: $S\ currentService \leftarrow null$
3: $Stack\ stack \leftarrow\ new\ Stack(m_0)$
4: **while** $stack \neq \emptyset$ **do**
5: `M` $m \leftarrow stack$.POP()
6: t.ADD(m)
7: **for all** $m_i \in m.edges$ **do** $stack$.PUSH(m_i)
8: **if** $m.edges = \emptyset$ **then**
9: **for all** $id_x \in t$ **do**
10: **if** $h(id_x) = E$ **then** CONTINUE
11: **if** $m_x.isAccessorMutator$ = `false` **then**
12: **if** $\left(\Delta_{id_{x-1}} h < 0 \;\&\&\; \Delta_{id_{x+1}} h > 0\right) \;||\; \left(\Delta_{id_{x-1}} h > 0 \;\&\&\; \Delta_{id_{x+1}} h \geq 0\right)$ **then**
13: $OP\ op \leftarrow new\ OP(m_x, h(id_x))$
14: $currentService \leftarrow new\ S().si$.ADD($op$)
15: $currentService.simpl.internalM$.ADD($m_x$)
16: $sList$.ADD($currentService$)
17: **if** $\left(\Delta_{id_{x-1}} h < 0 \;\&\&\; \Delta_{id_{x+1}} h > 0\right) \;||\; \left(h(id_{x-1}) = h(id_x)\right)$ **then**
18: **if** $op_x \in sList.s_j.si$ **then**
19: $op_x.inv \leftarrow h(id_x)$
20: $currentService.simpl.externalOP$.ADD($op_x$)
21: **else** $currentService.simpl.internalM$.ADD($m_x$)
22: $t \leftarrow \emptyset$
23: **end while**

vices of Fig. 2 (a). The service usage-relatedness values are given in the symmetric matrix of Fig. 3 (b), based on which the algorithm merges the pairs of singleton services, (s_3, s_5) and (s_4, s_6), since they have the highest usage-relatedness (values in bold). The algorithm does not merge more services, since their usage-relatedness values are lower than the proximity 0.2 that was used (the algorithm achieves its max effectiveness for this value in the running example (see Section 7)).

7 Experimental Evaluation

We implemented the vertical and horizontal mechanisms in Java (our research prototype is available online[13]) to evaluate them on two realistic Java case-studies[14], designed for organizations (e.g. universities) that have a high number of clients[15].

[13] ecs.victoria.ac.nz/foswiki/pub/Main/DionysisAthanasopoulos/migration.jar
[14] ecs.victoria.ac.nz/foswiki/pub/Main/DionysisAthanasopoulos/case-studies.zip
[15] For experimental purposes, we monitored the usage of the case-studies by 50 end-users.

Algorithm 2 Horizontal service-identification

Input: `List<S>` $sList$, `double` $threshold$
Output: `List<S>` $services$

1: **for all** $(s_i,\ s_j) \in sList \times sList$ **do**
2: $\quad r[i][j] \leftarrow \frac{MIN(s_i.si.op.inv, s_j.si.op.inv)}{MAX(s_i.si.op.inv, s_j.si.op.inv)} * \frac{|s_i.simpl.internalM \cap s_j.simpl.internalM|}{MAX(|s_i.simpl.internalM|, |s_j.simpl.internalM|)}$
3: **for all** $s_i \in sList$ **do** $services$.ADD(s_i)
4: **repeat**
5: $\quad (min,\ s_i,\ s_j) \leftarrow$ FINDMINPROXIMITYPAIR$(r, services)$
6: $\quad$ **if** $min \geq threshold$ **then**
7: $\quad\quad$ `S` $s \leftarrow$ MERGE(s_i, s_j)
8: $\quad\quad$ $services$.REMOVE(s_i, s_j)
9: $\quad\quad$ $services$.INSERT(s)
10: $\quad\quad$ UPDATE$(r, services, s)$
11: **until** $|services| = 1$ or $min < threshold$

Experimental setup. The first case-study is the small-sized `S3 app` of our running example (8 classes and 22 methods). The second case-study is a medium-sized information system (`IS`) that manages and stores information about students in a MySQL database (30 classes and 95 methods)[16]. `IS` offers the retrieval of student profile, the student insertion/update, and the insertion of end-users. The end-users of `IS` are students, administration employees, and system curators. We initially monitored the usage of the case-studies. In response to every usage-trace, we executed our mechanisms for a range of proximity values. When no more usage traces were available, we checked the conformance of the identified services to SRP and ISP using the following metrics.

Concerning SRP, we calculated the closeness of the identified services to the expected (externally reusable) services that we defined based on the system usage (e.g. see Fig. 2 (a)). To quantify the closeness, we assessed the precision and robustness of our approach (i.e. how many operations are and are not correctly grouped). The *F-measure* metric [21] has been widely used for jointly assessing precision and robustness/recall (Eq. 6). High F-measure indicates high closeness degree. We define the precision (resp. recall) as the percentage of the operations, which correctly participate in the identified services (true positives) over all the operations of the identified (resp. expected) services (true positives and false positives (resp. negatives)).

$$F - measure := \frac{2 * pr * r}{pr + r} \ \Big|\ pr = \frac{tp}{tp + fp} \text{ and } r = \frac{tp}{tp + fn} \tag{6}$$

Regarding ISP, we measured the interface usage-cohesion (IUC) of each identified service via using the metric in [18], which assumes that client groups are known beforehand. The higher the IUC, the higher the usage cohesion is. Since the metric assesses the IUC of front-end services, we extend it to measure the IUC of both front- and back-end services (see Eq. 7). We also calculated the average and variability (a.k.a. standard

[16] The recent approach in [17] uses an analogous number and scale of case-studies.

Table 4. The services identified by the mechanisms of our process for the running example.

(a) Vertical mechanism

S	SI	SIMPL	
		externalOP	internalM
s_1	op_3	$\{op_4\}$	$\{m_3\}$
s_2	op_4	$\emptyset$	$\{m_4, m_{10}, m_{11}, m_{20}\}$
s_3	op_9	$\emptyset$	$\{m_9, m_{12}, m_{13}, m_{19}\}$
s_4	op_5	$\{op_4, op_6\}$	$\{m_5, m_{12}, m_{13}, m_{14}, m_{21}\}$
s_5	op_7	$\{op_4, op_9\}$	$\{m_7, m_{14}, m_{22}\}$
s_6	op_6	$\{op_4\}$	$\{m_6, m_{14}, m_{15}, m_{16}, m_{17}\}$
s_7	op_8	$\{op_4, op_9\}$	$\{m_8, m_{18}\}$

(b) Horizontal mechanism

S	SI	SIMPL	
		externalOP	internalM
s_1	op_3	$\{op_4\}$	$\{m_3\}$
s_2	op_4	$\emptyset$	$\{m_4, m_{10}, m_{11}, m_{20}\}$
s_3	op_9	$\emptyset$	$\{m_9, m_{12}, m_{13}, m_{19}\}$
s_4	op_5	$\{op_4, op_6\}$	$\{m_5, m_{12}, m_{13}, m_{14}, m_{21}\}$
	op_7	$\{op_4, op_9\}$	$\{m_7, m_{14}, m_{22}\}$
s_5	op_6	$\{op_4\}$	$\{m_6, m_{14}, m_{15}, m_{16}, m_{17}\}$
	op_8	$\{op_4, op_9\}$	$\{m_8, m_{18}\}$

deviation[17]) of the IUC values of all the identified services of each system.

$$IUC(i) := \frac{\sum_{j=1}^{|clients|} \frac{used\_methods(j,i)}{all\_methods(i)}}{|clients|} \quad \Big| \; j \text{ denotes a client of a } \textit{front-end} \text{ service } i \text{ or a } \textit{back-end} \text{ service } i \text{ for which there is a sequence of services whose front-end service is used by } j \tag{7}$$

Results of the SRP evaluation. The F-measure values of both systems are depicted in Fig. 6. The figure also plots the percentages of the identified services (by dividing them with the total number of singleton services). We observe that our approach conforms to SRP with respect to the expected services in the `S3 app`, since it achieves the max F-measure for the proximity 0.2 (Fig. 6 (a)). In detail, the F-measure increases starting from a low value, then is maximized, and finally decreases until a medium value. To justify this behaviour, we check the ratios of the identified services. When the proximity is low, the ratios are also low, since many services are related and are finally merged (a few fat service interfaces are identified). When the proximity increases, the number of the identified services also increases (many thin and/or singleton service-interfaces are identified) and hence, the F-measure increases. From a proximity value and then, the F-measure decreases, since the service ratios increase. In the case of `IS`, our approach achieves high F-measure 0.71 for the proximity 0.3 (Fig. 6 (b)), which means that it conforms at a high degree to SRP. The achieved F-measure is not maximum, since our approach identifies less services (≤ 10) than the 14 expected services. The reason behind it is that our representation model breaks formed cycles. Thus, our approach misses back-end services that are reused via forming invocation cycles.

Results of the ISP evaluation. The average and standard deviation (`stdev`) of IUC values of both systems are depicted in Fig. 7. We observe that our approach conforms to ISP with respect to the expected services in the `S3 app`, since it achieves the max (resp. min) average (resp. `stdev` of) IUC value for the proximity 0.2 (Fig. 7 (a)). Note that the average IUC remains max for the proximity that is greater than 0.4 when only singleton service-interfaces are identified, since as mentioned in Section 2, IUC considers that systems with singleton services have maximum cohesion. In the case of `IS`, we observe that our approach achieves max (resp. min) average (resp. `stdev` of) IUC value for the proximity 0.3–1.0 (Fig. 7 (b)), conforming to ISP. Even if our approach identifies less than the expected services, its average IUC is maximum since it succeeds

[17] It equals to zero if the IUC values are the same and increases if the values are diverse.

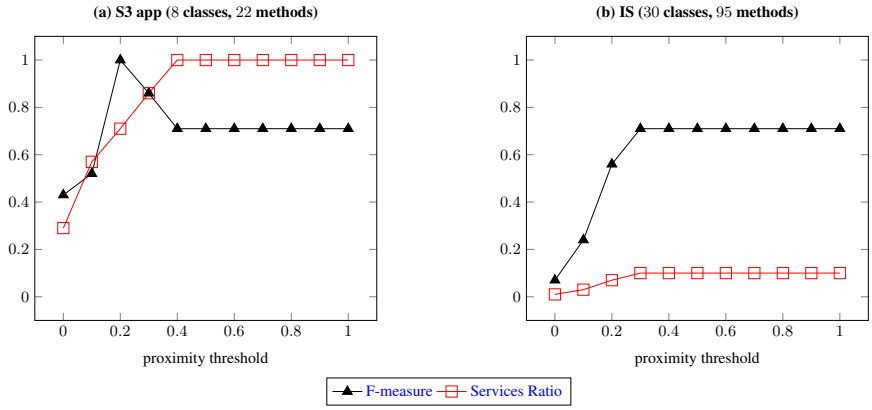


Fig. 6. The results of the SRP evaluation of our approach.

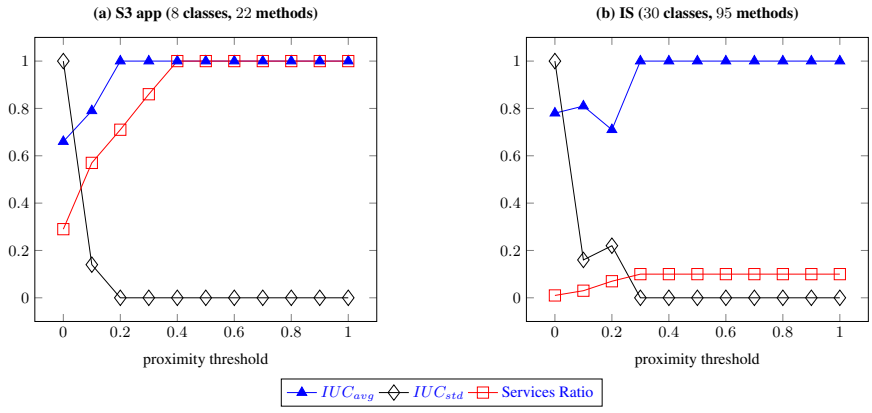


Fig. 7. The results of the ISP evaluation of our approach.

in identifying the front-end services for the four main system functionalities, along with their dependent back-end services, used by different client groups.

8 Conclusions and Future Work

We proposed the initial version of an online process that identifies services based on the diverse and varying external usage of OO systems. The evaluation results of the process on realistic case-studies show high effectiveness on identifying services that conform at a high extent to the fundamental design principles that are related to external usage.

Our future research purpose is to extend the process with more service-invocation patterns and ways to relate usage-traces. We also aim at extending the process to cope with invocation cycles. We further aim at conducting large-scale experiments using legacy systems. Finally, an interesting direction is the release of SoA systems that continue to evolve, re-identifying, re-encapsulating, and re-deploying candidate services.

References

1. T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.
2. M. Razavian and P. Lago. A systematic literature review on SOA migration. *Journal of Software: Evolution and Process*, 27(5):337–372, 2015.
3. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.
4. D. Romano, S. Raemaekers, and M. Pinzger. Refactoring fat interfaces using a genetic algorithm. In *International Conference on Software Maintenance & Evolution*, 2014.
5. S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. In *Working Conference on Reverse Engineering*, pages 115–124, 2006.
6. Z. Zhang, R. Liu, and H. Yang. Service identification and packaging in service oriented reengineering. In *International Conference on Software Engineering and Knowledge Engineering*, pages 620–625, 2005.
7. G. A. Lewis, E. J. Morris, D. B. Smith, and L. O'Brien. Service-oriented migration and reuse technique (SMART). In *International Workshop on Software Technology and Engineering Practice*, pages 222–229, 2005.
8. L. O'Brien, D. B. Smith, and G. A. Lewis. Supporting migration to services using software architecture reconstruction. In *International Workshop on Software Technology and Engineering Practice*, pages 81–91, 2005.
9. H. M. Sneed. Integrating legacy software into a service oriented architecture. In *European Conference on Software Maintenance and Reengineering*, pages 3–14, 2006.
10. R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage. A method engineering based legacy to SOA migration method. In *International Conference on Software Maintenance*, pages 163–172, 2011.
11. R. Khadka, A. Saeidi, S. Jansen, and J. Hage. A structured legacy to SOA migration process and its evaluation in practice. In *International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 2–11, 2013.
12. S. Alahmari, E. Zaluska, and D.De Roure. A service identification framework for legacy system migration into SOA. In *International Conference on Services Computing*, pages 614–617, 2010.
13. G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. Migrating interactive legacy systems to web services. In *European Conference on Software Maintenance and Reengineering*, pages 24–36, 2006.
14. F. Chen, Z. Zhang, J. Li, J. Kang, and H. Yang. Service identification via ontology mapping. In *International Computer Software and Applications Conference*, pages 486–491, 2009.
15. L. Aversano, L. Cerulo, and C. Palumbo. Mining candidate web services from legacy code. In *International Symposium on Web Systems Evolution*, pages 37–40, 2008.
16. Z. Zhang and H. Yang. Incubating services in legacy systems for architectural migration. In *Asia-Pacific Software Engineering Conference*, pages 196–203, 2004.
17. S. Adjoyan, A.-D. Seriai, and A. Shatnawi. Service identification based on quality metrics. In *International Conference on Software Engineering & Knowledge Engineering*, 2014.
18. M. Perepletchikov, C. Ryan, and Z. Tari. The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3(2):89–103, 2010.
19. K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2002.
20. O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.
21. R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.