A Type-Sensitive Service Identification Approach for Legacy-to-SOA Migration

Manel Abdellatif[1,2], Rafik Tighilt[2], Naouel Moha[2], Hafedh Mili[2], Ghizlane El Boussaidi[3], Jean Privat[2], and Yann-Gaël Guéhéneuc[4]

[1] Polytechnique Montréal, Montreal, Quebec, Canada
[2] Université du Québec à Montréal, Montreal, Quebec, Canada
[3] École de Technologie Supérieure, Montreal, Quebec, Canada
[4] Concordia University, Montreal, Quebec, Canada

Abstract. A common strategy for modernizing legacy systems is to migrate them to *service-oriented architecture* (SOA). A key step in the migration process is the identification of reusable functionalities in the system that qualify as *candidate services* in the target architecture. We propose *ServiceMiner*, a bottom-up service identification approach that relies on source code analysis, because other sources of information may be unavailable or out of sync with the actual code. Our *bottom-up, code-based* approach uses *service-type specific functional-clustering criteria*. We use a categorization of service types that builds on published service taxonomies and describes the code-level patterns characterizing types of services. We evaluate *ServiceMiner* on an open-source, enterprise-scale legacy ERP system and compare our results to those of two state-of-the-art approaches. We show that ServiceMiner automates one of the main labor-intensive steps for migrating legacy systems to SOA. It identifies architecturally-significant services with 77.9% of precision, 66.4% of recall, and 71.7% of F-measure. Also, we show that it could be used to assist practitioners in the identification of candidate services in existing systems and thus to support the migration process of legacy systems to SOA.

Keywords: Service Identification, Service types, Legacy Migration, Software Reuse

1 Introduction

The maintenance and migration of legacy software systems are central IT activities in many organizations in which these systems are mission-critical. These systems embed hidden knowledge that is of significant values. They cannot be simply removed or replaced because they execute effectively and accurately critical and complex business logic. However, legacy software systems are difficult to maintain and scale because their software and hardware become obsolete [1]. They must be modernized to ease their maintenance and evolution.

A common strategy for modernizing such systems is their migration to *service-oriented architecture* (SOA), which defines a style where systems are made of services that are reusable, distributed, relatively independent, and often heterogeneous [2]. Service Identification (SI) is considered one of the most challenging steps of the migration process [3]. It consists in identifying reusable groupings—clusters of functionalities in the legacy system that qualify as *candidate services* in the target architecture. Several SI approaches have been proposed in the literature [4,5,6,7,8,9,10]. However most of them have limited identification accuracy and usually require several types of inputs (e.g., business process models, use cases, activity diagrams, etc.) that may not be always available especially in the context of legacy systems. We argue that service identification should depend on service types to improve the identification accuracy by narrowing the search space through the types and their associated code-patterns. Service types can be used to classify service candidates according to a hierarchical-layered schema and offers the possibility to prioritize the identification of specific types of services according to the business requirements of the migration process. Also, in our prior work [11], we reported that several practitioners highlighted the importance of identifying service types when migrating legacy systems to SOA. They claimed that type-aware SI provides important information on the nature and business capabilities of the identified services. Besides, existing source-code SI approaches use similar *functional-clustering criteria*—typically *cohesion* and *coupling*, which lead to candidate services that are often architecturally irrelevant for the new SOA-based system.

Consequently, we propose *ServiceMiner*, a type-aware SI approach to support the migration of legacy systems to SOA. We consider a bottom-up approach relying on source code analysis, as other sources of information (e.g., business process models, use cases, activity diagrams, etc.) may be unavailable or out of sync with the actual code. We use a categorization of service types based on previous service taxonomies and describe the code-level patterns characterizing each type of service. We evaluate *ServiceMiner* on an open-source, enterprise-scale legacy ERP system and compare its results to those of two state-of-the-art SI approaches [7,5]. We show that our approach automates the identification of specific types of candidate services, which are architecturally significant for the new SOA-based system.

This paper is structured as follows. Section 2 presents the related work and describes the taxonomy of service types. Section 3 details the service identification approach. Section 4 presents the experimental validation of our approach and details the obtained results. We discuss in Section 5 our threats to validity and provide our recommendations. Finally, we conclude in Section 6 with future work.

2 Background and Related Work

We describe in this section related work considering service types, their limitations, and the taxonomies on which we build our approach.

2.1 Related Work

Several approaches were presented to identify services from legacy software. However, only five approaches [12,13,14,15,16] considered the types of services.

Marchetto *et al.* [13] proposed a stepwise type-sensitive service identification approach that extracts reusable services from legacy systems based on dynamic analysis of java-based systems. They proposed guidelines to identify Utility, Entity and Task services from legacy systems. They executed several test scenarios and extracted reusable functional groupings that they qualified as candidate services. They identified Utility services by manually mining non-business-centric functionalities and looking at cross-cutting functionalities that can be grouped and exposed as candidate Utility services. They extracted candidate Entity services by analyzing the persistent objects and the classes using them. Finally, they considered each main functionality of the target application as a possible candidate Task service. Although the proposed SI approach is type-sensitive, the identification is still manual and based on executing several test scenarios that may not cover all the functionalities of the system. The approach was validated on small Java systems, limiting its application on real enterprise systems.

Huergo *et al.* [15] proposed a method to identify services based on their types. They rely on UML class diagrams of object-oriented based systems from which they derive state machine diagrams to identify the states of the objects in the system. They start by manually identifying *Master data* that they define as entity classes considered to play a key role in the operation of a business. Each Master data is considered as a candidate Entity service. Next, they derive state machine diagrams that are related to the identified Master data. They analyze the transitions on the state machine diagrams and identify Task and Process services. The identification process is also not fully automated and relies on the manual identification of master data in the system that qualify as Entity services.

Alahmari *et al.* [12] identified services based on analyzing business process models. These business process models are derived from questionnaires, interviews and available documentations that provide atomic business processes and entities on the one hand, and activity diagrams that provide primitive functionalities, on the other hand. Different service granularity are distinguished in relation to atomic business processes and entities. Dependent atomic processes as well as the related entities are grouped together at the same service to maximize the cohesion and minimize the coupling. However the implementation details of the approach is not fully described.

Fuhr *et al.* [14] considered three types of services: Business, Entity, and Utility services, which are identified from legacy code based on a dynamic analysis technique. The authors relied on a business process model to identify related classes. Each activity in the business process model is executed and classes called during the execution of an activity are considered related. The identification of services is based on a clustering technique where the similarity measurement is the number of classes used together in one activity. The identified clusters are then manually mapped into the different service types. A strong assumption of this approach is that business process models are available to execute activities.

Grieger *et al.* [16] presented an approach identifying three service types when analyzing legacy code. The first type refers to initial Design services that implement business values. These occurrences are identified based on refining the existing legacy code related to business values. The second type corresponds to coarse-grained services, e.g., business processes. These are identified based on orchestrating other services related to the same underlying business process (i.e., structural dependent services). The last service type is related to services that implement crosscutting concerns and technical functionalities used across different services (i.e., Utility services). The identification of these services is based on partitioning the functionalities of multiple services to recover individual and common parts. The authors relied on a clone detection algorithm to extract cloned functionalities shared among different services. The identified cloned functionalities are given to software architects to decide if they should to be moved into an existing service or merged into a new one. The proposed approach highly depend on the manual and iterative refinement of the identified candidate services with software architects. The approach also lacks of empirical evidence on its reliability to support software architects during the migration process as there is no information about the quality of the identified services.

We notice that there is a lack of SI approaches that are type-sensitive. These approaches focus on identifying Business, Entity, and Utility services. Most of these approaches are not fully-automated and need other inputs than the source code (e.g., business processes, execution traces, state machine diagrams, etc.) to identify services in legacy systems.

Also, a number of primary studies have been proposed in the literature about SI without taking into account service types. Many of the proposed techniques rely on Business Process Models (BPMs), to identify services within the context of legacy migration [17,18,19]. These techniques decompose processes into tasks and then map these tasks to legacy source code elements to identify candidate services. Other SI techniques use heuristics based on the *technical properties* of services, as reflected in various metrics [5,8,7,20,21]. Such techniques often use these metrics to drive clustering and machine learning algorithms that identify software artifact clusters as candidate services. Other AI-based techniques use ontologies and Formal Concept Analysis to identify services in legacy systems [22,9,10]. However these techniques are too complex and not ready for industrial applications. There are also wrapping-based SI techniques that put service interfaces around *existing* functional components and subsystems [4,6,23], which solve integration problems but do not solve maintenance issues.

2.2 Taxonomy of Service Types

A number of service type taxonomies exists [2,12,13,14,15,16,24,25] that consider different aspects (e.g., domain specificity, granularity, governance) to distinguish service types. We study and combine these taxonomies and limit ourselves to those services types that are *distinguishable at the code level*. We distinguish between *domain-specific services*, and *domain-neutral services*. *Domain-specific Services* fall into four major types:

1. **Business services:** They correspond to business processes or use cases. These are services used by users. These services generally compose/use the Enterprise-task, Application-task, and Entity services described in the following.
2. **Enterprise services:** (Also called capabilities [24]) they are of finer granularity than business process services. They implement generic business functionalities reused across different applications.
3. **Application services:** These services provide functionalities specific to one application. They exist to support reuse within one application or to enable business process services [24].
4. **Entity services:** (Also called information or data services) they provide access to and management of the persistent data of legacy software systems. They support actions on data (CRUD) and may have side-effects (i.e., they modify shared data).

Domain-neutral Services are services that provide functionalities to develop, use, and compose domain-specific services:

1. **Utility services:** They provide some cross-cutting functionalities required by domain-specific services. Logging and authentication services are examples of Utility services.
2. **Infrastructure services:** They allow users deploying and running SOA systems. They include services for communication routing, protocol conversion, message processing and transformation. They are sometimes provided by an Enterprise Service Bus (ESB).

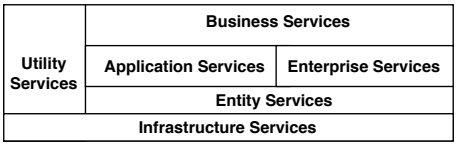

Fig. 1. Taxonomy of service types

In our prior work [11], we validated this taxonomy through an industrial survey with practitioners who participated in real migration projects to SOA. None of them mentioned the identification of other types of services during the migration process. In the following, we will consider the identification of only Utility, Entity and Application services and detail how we can identify such types of services through the analysis of the source code of legacy software systems. In fact, non service-oriented legacy systems are likely not to contain *infrastructure services* and, thus, we do not consider this type of services in our analysis. Second, we do not distinguish between *Application services* and *Enterprise services* because they only differ in terms of scope of reuse: within a single system vs. across systems. Also, we do not consider *Business services* because (1) they orchestrate other services, such as *Enterprise* and *Application services*, and (2) other sources of information, e.g., business process models, are required to detect them.

3 Service Identification by Type: Our Approach

Figure 2 summarizes our SI approach, *ServiceMiner*, which consists of two phases: (1) a *pre-processing phase* in which we build the call graph of the system based on source code analyses, perform an initial clustering of highly connected classes, and compute the code metrics used in the services detection rules and (2) a *processing phase* in which we apply rules on the generated clusters to filtrate, reorganize, and classify them to identify candidate services and their types.

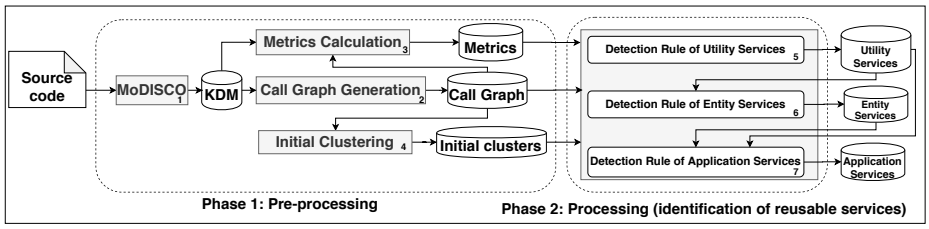

Fig. 2. Overview of *ServiceMiner*

3.1 Pre-processing Phase

Call Graph Generation. Our SI rules in Table 1 use code metrics, such as fanin and fanout, computed on the *call graph* of the legacy system. Thus, in a first step, we parse the source code of the legacy system and build its call graph. Legacy systems come in different languages and may combine several technologies. The OMG *Knowledge Discovery Metamodel* (KDM) [26] was defined to represent (legacy) systems at different levels of abstraction, regardless of languages and technologies. Thus, we use MoDISCO [27], an Eclipse-based open-source implementation of the KDM that provides (1) an extensible parsing framework to obtain KDM models from files in different languages and (2) a framework to navigate the KDM models, which we extend to generate call graphs of legacy systems.

Metrics Calculation. Our rules use class-level and method-level metrics. We use the call graphs obtained in the previous step to compute class-level metrics. For the sake of simplifying the implementation of our SI approach, we use *Understand*[5] to compute method-level metrics. We also analyze the static relationships between the modules of the systems and assign a weight to each of them according to their relative importance. A module may be a procedure, an object, or any other piece of software depending on the programming language. A relationship may be *generalization*, *aggregation*, or *association*, between classes in object-oriented systems for example. The total relationship strength between a pair of related modules is:

[5] http://www.scitools.com

$$Weight(C_i, C_j) = \sum_{t=1}^{T} W_t \times NR_t$$

where C_i and C_j are modules, T is the number of relationships, W_t the weight assigned to a relationship type t, and NR_t the number of type t relationships between C_i and C_j. We study the relationships to ensure that only related modules are grouped together in the following steps.

Initial Clustering : Identification of Functional Groupings. The SI rules in Table 1 apply to candidate clusters that group functionalities in a legacy system. Finding such groupings within a call graph is akin to a call-graph clustering problem. We rely on Kruskal's maximum spanning-tree algorithm [28] to generate our initial set of clusters because (1) it is an efficient polynomial-time algorithm for generating clusters based on a graph structure, (2) it was used by several state-of-the-art SI approaches [7,29], and (3) a free implementation of the algorithm is provided in the open-source Java library *Jgrapht*, which can be easily integrated into our implementation. To enhance the clustering results of the spanning-tree algorithm, we put the modules that are reachable only from certain other modules in the same cluster. For example in case of object-oriented systems, if a class `A` is only accessible from a class `B` and the two classes are not in the same cluster then they must be grouped in same one.

3.2 Processing Phase: Identification of Reusable Services

Not all the clusters qualify as candidate services. In this step, we select the functional groupings/clusters to migrate while considering the different types of services. We first discuss service types and their code patterns qualitatively and then express them as rules (see Table 1).

Each type of service is mutually exclusive and their detection is hierarchical: first we detect candidate *Utility services*. Within the remaining groupings, we detect *Entity* and *Application services* as follows:

1. **Utility services:** They provide highly generic functionalities that are separate from domain-specific business processes and reusable across a range of business functionalities [2]. We detect Utility services by identifying groupings that satisfy the rule described in Table 1: high fanin (groupings that are highly solicited/called) and low fanout (groupings that do not call many other clusters). Utility services are *domain-neutral* so the identified groupings should not be persistent or contain database queries.
2. **Entity services:** They represent and manage *domain-specific* business entities, such as products, invoices. They are *data-centric* and reusable by other *domain-specific* services, such as Application services. We classify a grouping as an Entity service with (1) high fanin, (2) low fanout, (3) persistent modules, (4) access to the infrastructure (e.g., database), and (5) fine grained.

3. **Application services:** They are domain-specific and provide business functionalities specific to one system. They have low fanin compared to Entity and Utility services. They can also compose functionalities provided by Entity services. They generally perform complex computation. We use McCabe complexity metric as well as error handling capabilities to measure complexity computation. We classify a grouping as an Application service if it has (1) a call to at least one Entity service, (2) a high McCabe complexity, and–or (3) an error handling.

Service Type	**Detection rules**
Utility Services	Very High Fanin **AND** Very Low Fanout **AND Not** persistent
Entity Services	**Not** Utility service **AND** High Fanin **AND** Low Fanout **AND** Persistent **AND** Access to infrastructure **AND** Fine grained
Application Services	**Not** Utility **AND Not** Entity **AND** Low Fanin **AND** (Call to Entity $\geq$1 **OR** High McCabe Complexity **OR** Error Handling)

Table 1. Detection rules of services according to their types

4 Experimental Validation

Our validation divides into (1) a quantitative validation of *ServiceMiner* on a case study in comparison to a ground-truth, (2) a qualitative validation of the identified services that are related to a particular feature of our case study, and (3) a comparison of our identification results with those of two other state-of-the-art approaches [5,7].

4.1 Case Study

As case study, we choose *Compiere* because it is one of the few available, large, and open-source *legacy* system available on the Internet. Compiere is a *legacy system* because it is a large ERP system with a long history, first introduced by *Aptean* in 2003[6]. It provides businesses, government agencies, and non-profit organizations with flexible and low-cost ERP features[7], such as business partners management, monitoring and analysis of business performance, control of manufacturing operations, warehouse management (automating logistics), purchasing (automating procurement to payment), materials management (inventory receipts, shipments, etc.), and sales order management (quotes, book orders, etc.). It supports different databases, such as Oracle and PostgreSQL. We use Compiere v3.3 because (1) it is the first stable release of the system, (2) it was released more than 15 years ago, (3) it includes 2,716 classes for more than 530 KLOC, and (4) it is not service-oriented.

[6] http://www.aptean.com
[7] http://www.compiere.com/products/capabilities/

4.2 Ground-Truth Architecture

We need a *ground-truth* service-oriented architecture of Compiere to assess our approach. We asked two independent Ph.D. and Master's students to identify services in Compiere. They relied on several artifacts to build manually the ground-truth architecture by (1) analyzing the system, (2) understanding it, and (3) extracting its reusable parts that could become services. They used *Understand* to recover its design and to visualize class dependencies. They also generated views of its call graph that we make available online[8]. They also reviewed extensively the system documentation as well as its source code to have the best possible understanding and accurately identify services that can be integrated in the targeted SOA-based system. They found 473 services, which they annotated manually according to their type.

4.3 Quantitative Validation

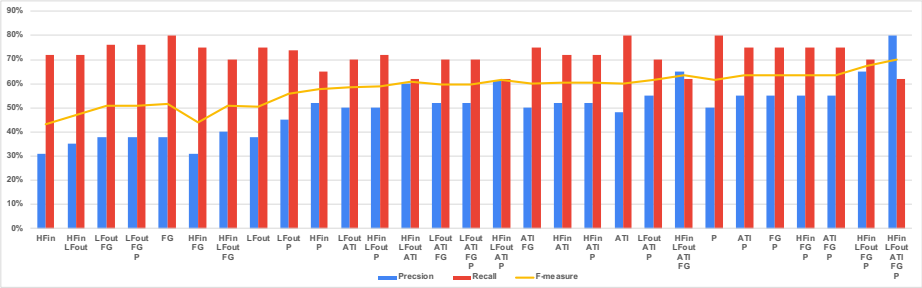

Fig. 3. Evaluation of the detection rules of Entity Services

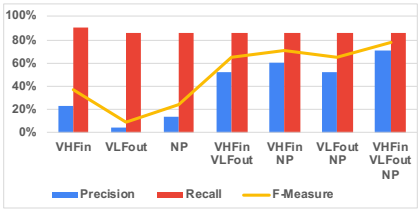

Fig. 4. Evaluation of the detection rules of Utility Services

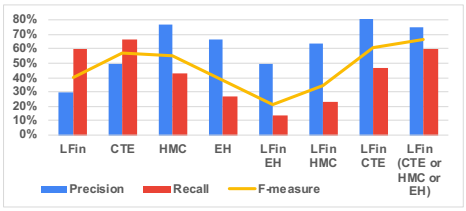

Fig. 5. Evaluation of the detection rules of Application Services

We applied *ServiceMiner* on Compiere to show its practical accuracy in identifying services in an existing system by considering, for each service type, several combinations of the criteria related to each rule. We measured precision, recall, and F-measure for each rule and report the results in Figures 3, 4, and 5.

[8] http://si-serviceminer.com

For example, for Utility services, we considered several possible combinations of the criteria, such as *"very high fanin"*, *"very low fanout"*, and *"Not persistent"*. We tested all possible combinations of these criteria and measured precision, recall, and F-measure for each rule. As shown by Figure 4, the best F-measure is obtained when considering the three criteria to identify such type of services: considering clusters with only very high fanin or very low fanout or clusters that are only not persistent leads to poor precision values.

We did the same evaluation process to study the effectiveness of the detection rules for Entity and Application services. As shown by Figures 3, 4, and 5, the best F-measure values were obtained when applying all the detection rules detailed in Section 3.2, which we used in *ServiceMiner* to identify in Compiere 403 services: 24 Application services, 278 Entity services, and 101 Utility services. We report in Table 2 the accuracy of *ServiceMiner*: a precision of 77.9%, a recall of 66.4%, and a F-measure of 71.7%.

Table 2 shows that the best accuracy of *ServiceMiner* pertained to Utility services with a precision of 77.9% and a recall of 86%. The identified Utility services relate to logging, Web uploading, printing, etc. For Entity services, we obtained a precision of 80.2%, a recall of 62.3%, and a F-measure of 70.1% with services for products, orders, invoices, etc. We missed some Entity services that have a low fanin/a high fanout because of our choice of metrics and thresholds, which could be refined by the developers when applied to their own systems. We observed a precision of 75%, a recall of 60%, and a F-measure of 66.7% for Application services, which relate to payment processing, tax calculation, and inventory management. The identification of Application services depends on the previous identifications of Entity and Utility services and, thus, false-positive Application services were mainly due to some Entity and Utility services being incorrectly labeled as Application services, such as caching-related services and Web-project deployment services.

Although we missed the identification of some services, we reduced the developers' effort needed to identify services by avoiding the manual identification of at least 66% of the candidate reusable services. Our recall could be improved by setting different thresholds and iterating through the identification process.

Service Type	**# Services**	**Precision**	**Recall**	**F-measure**
Application	24	(18/24) 75.0%	(18/30) 60.0%	66.7%
Entity	278	(223/278) 80.2%	(223/358) 62.3%	70.1%
Utility	101	(73/101) 72.3%	(73/85) 86.0%	78.6%
Total	403	(314/403) 77.9%	(314/473) 66.4%	71.7%

Table 2. Overview of Service Identification Accuracy with *ServiceMiner*

4.4 Qualitative Validation

We apply *ServiceMiner* on Compiere to identify relevant services in the system. We take the example of the sales orders management in Compiere and detail how *ServiceMiner* helps practitioners identify services related to this feature.

Sales orders management entails quotations, sales orders, and invoicing, linked to the shipment of goods to customers. The initial clustering step of our approach builds a set of candidate clusters that we then filtrate with our detection rules to identify candidate services. First, we identify Utility services by applying the first rule in Table 1, which yields Utility services about logging and printing. These services provide cross-cutting functionalities called by multiple other services (e.g., sales) with very low fanout and no persistence.

Second, we identify Entity services, i.e., clusters representing business entities. They refer to the business entities of the system, such as products, invoices, business partners, warehouses, and bank statements. These entities are persistent, have access to the database, and are invoked by other domain-specific services (e.g., Application and Business services).

Third, we apply our last detection rule in Table 1 to identify Application services among the remaining clusters. An example of Application service related to sales orders processing is the payment service responsible for generating payments of the orders based on the information provided by the invoice, business partner, and bank statement Entity services. It is also responsible for handling errors when the payment is unsuccessful.

We obtained architecturally significant candidate services thanks to the application of our type-aware service identification approach on Compiere. We believe that it can assist practitioners in the identification of candidate services because it automates the SI process of Utility, Entity, and Application services with acceptable precision and recall.

4.5 Comparison with State-of-the-art Approaches

We chose two existing approaches to compare their results against those of *ServiceMiner*: *MOGA-WSI* [7] and *Service Cutter* [5]. These two were the only available approaches. *MOGA-WSI* uses spanning trees and provides candidate services with different levels of inter-service coupling. It relies on genetic and multi-objective optimization algorithms to refine an initial set of services. It also considers a set of managerial goals, such as cost effectiveness, ease of assembly, customization, reusability, and maintainability. *Service Cutter* is a graph-based approach considering 16 coupling criteria and two kinds of clustering algorithms, *Girvan-Newman* (GN) [30] and *Epidemic Label Propagation* (ELP) [31], which differ in terms of their (non-)deterministic behavior.

We assess our approach with respect to a ground-truth architecture and in comparison to the two tools using measures of clustering and information retrieval: MojoFM [32], *Architecture2Architecture* (A2A) [33], precision, and recall. We rely on these metrics to study the identification results of each approach regardless of the types of services.

Table 3 lists the identification results and shows that our approach outperforms *MOGA-WSI* and *Service Cutter* for all the reported metrics. We tried several configurations for both tools but all showed poor results in comparison to our approach. We observed that these approaches generate very unbalanced services. For example, *MOGA-WSI* identified a service of 253 classes and 143

services of one to six classes. Similarly, *Service Cutter* (EPL) identified two coarse-grain services and 393 fine-grained ones. Although service identification using *Service Cutter* with *Girvan-Newman* is deterministic, we were limited to a maximum number of 30 services, which lead to poor identification results.

We argue that our SI approach outperforms the two other approaches because: (1) *ServiceMiner* follows a stepwise process, which identifies Utility services then Entity services and finally Application services, (2) it uses simple, straightforward metrics instead of complex goals, like maintainability, which are subjective and more difficult to define and measure, and (3) the studied state-of-the-art tools are proof of concepts, which may limit their applicability on enterprise-scale systems, such as Compiere.

Approach	**#Services**	**MojoFM**	**A2A**	**Precision**	**Recall**	**F-measure**
MOGA-WSI	396	11.0%	42.0%	14.0%	13.0%	13.5%
Service Cutter (EPL)	395	15.7%	51.0%	12.2%	10.3%	11.2%
Service Cutter (GN)	30	21.6%	41.0%	15.6%	9.7%	14.1%
ServiceMiner	**403**	**65.0%**	**73.0%**	**77.9%**	**66.4%**	**71.7%**

Table 3. Comparison results of service identification approaches

5 Discussion on the Approach

Threats to Validity. Construct validity concerns the accuracy of some observations with respect to a theory. In our validation process, we should have relied on a real post-migration SOA-based system. However, we could not find any open-source enterprise-scaled system that was migrated to SOA. Thus, we relied on a *ground-truth* architecture to validate quantitatively the services identified by our approach. We are aware that there is no single "correct" SOA for a given legacy system. However, we relied on several artifacts (e.g., official documentations, source code analysis, etc.) and studied in-depth the system to identify reusable sets of classes that could be packaged into services. Also, we share our *ground-truth* architecture to allow others to confirm/infirm our claims. We put the original source code as well as the identified services online[9], which also reduces threats to reliability validity.

Our SI approach as well as its validation depend on several algorithms and thresholds that threaten the internal validity of our results. To mitigate these threats, (1) we implemented *MOGA-WSI* based on their original papers to the best of our understanding and shared it for investigation and replication[10]; (2) we used the best identification results of these tools to compare our approach; (3) we explored the use of different metrics and threshold values; and, (4) we chose the spanning-tree algorithm for its ease of use and available, open-source implementation. Future work should consider comparing with other algorithms to vet further the reliability of our approach in comparison to other SI approaches.

[9] http://si-serviceminer.com/ICSOC-2020-Replication
[10] https://github.com/MPoly2018/MOGA-WSI

We know that service detection rules may slightly differ from one system to another. However, our detection rules are easily customized, being flexible and extensible. We recommend to consider the same processing steps in the same order than in our approach to identify the services in existing systems according to their types. Also, legacy systems most likely embed poor design and coding practices, e.g., code smells, that reduce the separation of concerns within/between classes, which reduces the precision/recall of static-based SI approaches.

Our case study may not be representative of all legacy software systems, which limits the generalizability of our results but *Compiere* is large and complex enough to validate our approach while we continue our search for other large, open-source systems. Finally, the identified services may not be representative of all service types, which may also limit the generalizabiltiy of our results. Our approach is extensible to new service types. It can also be extended to identify microservices [34] by mapping each Utility and Entity service identified by *ServiceMiner* to a microservice and decomposing each identified Application service into smaller microservices that each have a single responsibility.

Discussions and recommendations. We believe that our approach is beneficial for both researchers and practitioners interested in migrating legacy systems to SOA because (1) we automate the SI process, which is one of the most labor-intensive step for migrating such systems to SOA; (2) our SI approach yields to architecturally significant candidate services that can be packaged and integrated in the targeted SOA-based system while identifying their types; (3) our approach offers the possibility to prioritize the identification of specific service types based on their importance and the architectural/business needs of the migration process; and, (4) our approach is extensible to new technologies/languages, thanks to its use of the KDM metamodel as intermediate representation for C, C++, Python, etc.

Finally, we recommend to consider service types to identify services in existing systems to improve accuracy. We also recommend to order the services to be migrated. We suggest to start first with Utility services because they are highly reusable and invoked by other services in the system (e.g., Entity, Application, Business process services, etc.); second, to continue with Entity services because they manage and represent the business entities of the system and are used by the other services; third, to identify Application services as they compose functionalities provided by Entity services; finally, to identify Business services that manage and compose/use the previous types of services.

6 Conclusion and Future Work

In this paper, we proposed *ServiceMiner*, a type-aware service identification (SI) approach for the migration of legacy software systems to SOA. *ServiceMiner* helps during the key step of identifying reusable service candidates using a taxonomy of service types. We evaluated *ServiceMiner* on a real-world legacy ERP system and compared its results with those of two state-of-the-art SI approaches. We showed that, in general, *ServiceMiner* identified relevant,

architecturally-significant services with 77.9% of precision, 66.4% of recall, and 71.7% of F-measure. Also, we showed that it outperformed the state-of-the-art SI approaches by providing more architecturally-significant services. We believe that *ServiceMiner* can thus be used to assist practitioners in the identification of candidate services in their systems.

As future work, we will consider the identification of other types of services, such as Enterprise and Business services. We will also perform a qualitative validation of the significance and relevance of the identified services with developers. We will compare other algorithms to further study the reliability of our approach. Finally, we will complete our SOA migration road-map by exploring automatic service packaging techniques to efficiently package and integrate the identified services into the targeted SOA platform.

References

1. G. Lewis, E. Morris, L. O'Brien, D. Smith, and L. Wrage, "Smart: The service-oriented migration and reuse technique," DTIC Document, Tech. Rep., 2005.
2. T. Erl, *SOA Principles of Service Design*. NJ, USA: Prentice Hall PTR, 2007.
3. R. Khadka, A. Saeidi, S. Jansen, and J. Hage, "A structured legacy to soa migration process and its evaluation in practice," in *MESOCA*, 2013, pp. 2–11.
4. G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "Migrating interactive legacy systems to web services," in *CSMR*, 2006, p. 10.
5. M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *ESOCC*, 2016, pp. 185–200.
6. R. Rodríguez-Echeverría, F. Maclas, V. M. Pavón, J. M. Conejero, and F. Sánchez-Figueroa, "Generating a rest service layer from a legacy system," in *Information System Development*. Springer, 2014, pp. 433–444.
7. H. Jain, H. Zhao, and N. R. Chinta, "A spanning tree based approach to identifying web services," *International Journal of Web Services Research*, vol. 1, no. 1, p. 1, 2004.
8. S. Adjoyan, A. Seriai, and A. Shatnawi, "Service identification based on quality metrics object-oriented legacy system migration towards SOA," in *SEKE*, 2014, pp. 1–6.
9. M. J. Amiri, S. Parsa, and A. M. Lajevardi, "Multifaceted service identification: Process, requirement and data," *ComSIS*, pp. 335–358, 2016.
10. Z. Zhang, H. Yang, and W. C. Chu, "Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration," in *QRS*, 2006, pp. 385–392.
11. M. Abdellatif, G. Hecht, H. Mili, G. Elboussaidi, N. Moha, A. Shatnawi, J. Privat, and Y.-G. Guéhéneuc, "State of the practice in service identification for soa migration in industry," in *ICSOC*. Springer, 2018, pp. 634–650.
12. S. Alahmari, E. Zaluska, and D. De Roure, "A service identification framework for legacy system migration into soa," in *SCC*. IEEE, 2010, pp. 614–617.
13. A. Marchetto and F. Ricca, "From objects to services: toward a stepwise migration approach for java applications," *STTT*, vol. 11, no. 6, p. 427, 2009.
14. A. Fuhr, T. Horn, and V. Riediger, "Using dynamic analysis and clustering for implementing services by reusing legacy code," in *WCRE*. IEEE, 2011, pp. 275–279.

15. R. S. Huergo, P. F. Pires, and F. C. Delicato, "Mdcsim: A method and a tool to identify services," *IT Convergence Practice*, vol. 2, no. 4, pp. 1–27, 2014.
16. M. Grieger, S. Sauer, and M. Klenke, "Architectural restructuring by semi-automatic clustering to facilitate migration towards a service-oriented architecture," *Softwaretechnik-Trends*, vol. 34, no. 2, 2014.
17. E. Souza, A. Moreira, and C. De Faveri, "An approach to align business and it perspectives during the soa services identification," in *ICCSA*, 2017, pp. 1–7.
18. H. M. Sneed, C. Verhoef, and S. H. Sneed, "Reusing existing object-oriented code as web services in a soa," in *MESOCA*. IEEE, 2013, pp. 31–39.
19. R. S. Huergo, P. F. Pires, and F. C. Delicato, "A method to identify services using master data and artifact-centric modeling approach," in *ACM SAC*, 2014, pp. 1225–1230.
20. A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. O. Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *ICSA*. IEEE, 2020, pp. 157–168.
21. I. Saidani, A. Ouni, M. W. Mkaouer, and A. Saied, "Towards automated microservices extraction using muti-objective evolutionary search," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 58–63.
22. M. Djeloul, "Locating services in legacy software:information retrieval techniques, ontology and fca based approach," *WSEAS Trans. Comput. (Greece)*, 2012.
23. G. Chenghao, W. Min, and Z. Xiaoming, "A wrapping approach and tool for migrating legacy components to web services," in *ICNDC*, 2010, pp. 94–98.
24. S. Cohen, "Ontology and taxonomy of services in a service-oriented architecture," *The Architecture Journal*, vol. 11, no. 11, pp. 30–35, 2007.
25. OpenGroup, "The open group soa reference architecture," [Online; accessed 1-June-2020].
26. G. E. Boussaidi, A. B. Belle, S. Vaucher, and H. Mili, "Reconstructing architectural views from legacy systems," in *WCRE*, 2012.
27. H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *IST*, vol. 56, no. 8, pp. 1012–1032, 2014.
28. J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.
29. G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.
30. M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.
31. U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.
32. Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.
33. J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 486–496.
34. S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.