ITC 1/48
Journal of Information Technology and Control
Vol. 48 / No. 1 / 2019
pp. 71-89
DOI 10.5755/j01.itc.48.1.21566

COBOL Systems Migration to SOA: Assessing Antipatterns and Complexity

Received 2018/09/04 | Accepted after revision 2018/12/19

http://dx.doi.org/10.5755/j01.itc.48.1.21566

COBOL Systems Migration to SOA: Assessing Antipatterns and Complexity

Cristian Mateos, Alejandro Zunino

ISISTAN Research Institute, UNICEN; Paraje Arroyo Seco Str., B7000, Tandil, Argentina; phone: +54 249 4385681; e-mails: {cristian.mateos,alejandro.zunino}@isistan.unicen.edu.ar

Also CONICET, The National Scientific and Technical Research Council; Godoy Cruz Str. 2290, C1425FQB, Buenos Aires, Argentina; phone: +5411 4899-5400

Andres Flores

GIISCo Research Group, Faculty of Informatics; National University of Comahue; Buenos Aires Str. 1400, 8300, Neuquén, Argentina; phone: +54 299 4490300 638; e-mails: andres.flores@fi.uncoma.edu.ar

Also CONICET, The National Scientific and Technical Research Council; Godoy Cruz Str. 2290, C1425FQB, Buenos Aires, Argentina; phone: +5411 4899-5400

Sanjay Misra

Atilim University, Kızılcaşar Mahallesi Str., 06836, Incek, Ankara, Turkey; phone +90 (312) 586 80 00

Also Covenant University; Km. 10 Idiroko Road, P.M.B 1023, Ota, Nigeria; phone: +234 01 7900724; e-mail: sanjay.misra@covenantuniversity.edu.ng

Corresponding author: cristian.mateos@isistan.unicen.edu.ar

SOA and Web Services allow users to easily expose business functions to build larger distributed systems. However, legacy systems – mostly in COBOL – are left aside unless applying a migration approach. The main approaches are direct and indirect migration. The former implies wrapping COBOL programs with a thin layer of a Web Service oriented language/platform. The latter needs reengineering COBOL functions to a modern language/platform. In our previous work, we presented an intermediate approach based on direct migration where developed Web Services are later refactored to improve the quality of their interfaces. Refactorings mainly capture good practices inherent to indirect migration. For this, antipatterns for WSDL documents (common bad practices) are detected to prevent issues related to WSDLs understanding and discoverability. In this paper,

we assess antipatterns of Web Services' WSDL documents generated upon the three migration approaches. In addition, generated Web Services' interfaces are measured in complexity to attend both comprehension and interoperability. We apply a metric suite (by Baski & Misra) to measure complexity on services interfaces – i.e., WSDL documents. Migrations of two real COBOL systems upon the three approaches were assessed on antipatterns evidences and the complexity level of the generated SOA frontiers – a total of 431 WSDL documents.

KEYWORDS: Legacy System Migration, Service-Oriented Architecture, Web Services, Direct Migration, Indirect Migration, WSDL Antipatterns, WSDL Complexity

1. Introduction

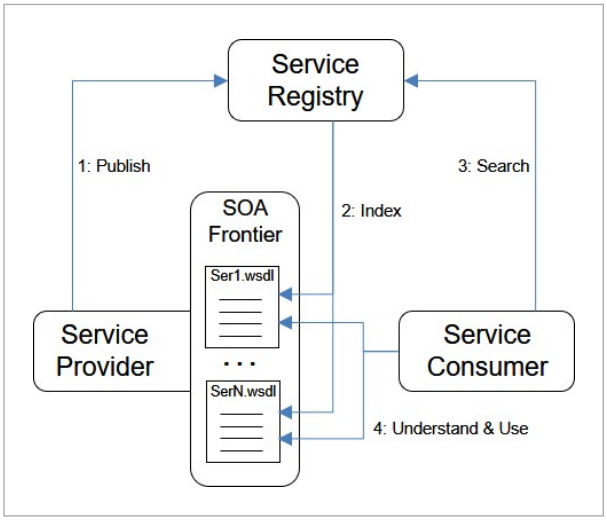
Organizations still relying on out-of-date supporting systems – e.g., in COBOL – are lately in the urgency to migrate towards new technologies such as Web 2.0, Web Services or mobile devices. The need is mainly driven to avoid high IT operational costs – e.g., mainframes – while increasing visibility to reach new markets [28, 34, 42]. System migration implies moving to new software environments/platforms while preserving legacy data and business functions [1]. Nowadays, a common architectural option is SOA (Service Oriented Architecture) [31], where systems are built from independent building blocks called services that can be invoked remotely. Services expose functionality that any system can use within/across the owner organization boundaries. The Web Services technology is the common way to materialize SOA, where services interfaces are described in WSDL (Web Service Description Language) [14]. Then, legacy to SOA migration mainly produces a SOA frontier, the set of WSDL documents describing the functionality of a service oriented system, as shown in Figure 1.

Two main approaches for migration to SOA are: direct and indirect migration [28, 29, 34]. The former represents a black-box (or bottom-up) approach that allows organizations to modernize their systems in a rapid and low cost manner. This is done by adding a new software layer to wrap the legacy functionality, which remains implemented with old technologies. The latter is a white-box (or top-down) approach that encompasses more elaborated reengineering concepts and techniques, and are often driven by properties of the desired service-based system from a (non) functional standpoint – leading to higher time/cost.

According to a recent study in [16], a common problem of SOA migration in many companies yields on prioritizing the technology perspective over a business one. The business process of the company is actually run and enforced by the legacy system [35]. A drastic change when migrating to SOA might affect such significant business value. In this sense, following an indirect migration might lead to a SOA system implementation that seriously impacts on business *a posteriori*. The new system architecture that might appear successful at that time could actually be not aligned with the company's business goals [16]. Therefore, both technical and business perspectives must be carefully attended to avoid such a mistake, causing a growing cost without return on investment.

In this sense, the key factor is the potential to reuse the legacy systems as components in SOA by exposing their functionality as services [16, 20] – the SOA frontier. This implies opening up the hidden business logic and properly concert it into new services – i.e., service abstraction. However, following a direct migration might not prevent the resulting SOA frontier from not fulfilling the new strategic business goal to-

Figure 1
SOA frontier and SOA roles



wards crossing the company boundaries into a global shared partnership. Here another key factor involves service discoverability that benefits the reuse of exposed legacy functions as services.

Attending these issues and key factors, in our previous work [31], we presented an intermediate approach based on direct migration in which developed Web Services are later improved on their interfaces quality – attending concerns inherent to an indirect approach. A quick migration is less costly but WSDL documents (the SOA frontier) might suffer from antipatterns – e.g., bad naming conventions and redundant operations [26] – affecting services readability and discoverability. Direct migration to SOA is often performed via a 1-to-1 mapping between legacy modules and Web Services. Hence, identifying and avoiding antipatterns is disregarded [30]. Indirect migration may address this issue but at a large investment in cost/time. The intermediate migration approach, called COB2SOA is focused on COBOL systems, and identifies refactoring opportunities to be applied on SOA frontiers. Through an automatic analysis of WSDL documents and COBOL files, antipatterns evidence detection is done. The antipatterns catalogue in [26] is applied, since it includes some of the antipatterns most frequently found in service-based systems [27]. After that, specific refactorings from the Fowler et al.'s catalogue [15] are identified as suggested remedy actions, to increase the quality level of WSDL documents.

All in all, organizations truly need a suitable approach to help migrating legacy systems to SOA, which is capable to ensure best quality of SOA frontiers. Hence, in this paper we address a specific quality concern related to the complexity of WSDL documents, which may seriously affect comprehension and interoperability. This may impact new business opportunities and partnerships, as services must be consumed from heterogeneous systems. Tightly closed functional constraints of data processing implemented in legacy modules are now openly exposed as operational data exchange protocols through WSDL documents. The use of XML (eXtensible Markup Language) documents serve to data message exchange. Certain message data structure definitions into WSDLs and XMLs might affect understandability.

Thereby, we make use of a recent metric suite (proposed by Baski & Misra [4]) to measure the complexity of services interfaces – i.e., WSDL documents. Measurable aspects entail the structure of requesting and responding messages of WSDLs operations. A factor that may increase complexity is the number of arguments within a message and their data types. Arguments can include built-in data types or complex data type structures. However, when similarly-structured complex types are defined, a familiarity factor arises that may decrease complexity. As such, this metric suite produces further trade-off information to be aware of design decisions about WSDLs. Concretely, we could prevent from interoperability problems by measuring the complexity that might be injected on a SOA frontier by following a migration approach.

Benefits and drawbacks of the three migration approaches are exposed through the migration of two real COBOL systems to SOA. According to surveys and studies by Gartner consulting, over 200 billion lines of COBOL code are still running worldwide [30]. From this experiment, a clear outcome is given about assessing both antipatterns and complexity metrics were performed in different stages during migration. The SOA frontiers produced by the three approaches involved 431 WSDL documents.

It is worth noting that this paper is an extension of our previous paper published in ICIST 2017 [22]. In this work, we extend such paper by a) significantly expanding the discussion of related works – i.e., direct and indirect migration approaches and proposals of a combined strategy; b) extending the description of the COB2SOA approach and its underpinnings so as to allow practitioners to materialize COB2SOA in supporting tools; c) evaluating the migration of two real COBOL systems upon three migration approaches – i.e., direct, indirect and COB2SOA – detailing both discovered evidences of antipatterns on generated WSDL documents and XSD models, and the achieved complexity level of the delivered SOA frontier. In addition, a quantitative analysis reveals LOC (Lines of Code), comments and number of offered operations from produced WSDL documents as well as the required migration times.

The rest of the paper is as follows. Section 2 presents preliminary concepts, including the antipatterns catalog and the complexity metric suite. Section 3 discusses relevant related work. Section 4 presents details of the COB2SOA approach. Section 5 describes the migration experiments performed. Conclusions and future work are presented afterwards.

2. Preliminary Concepts

As Web Services functionality is exposed through WSDL documents, their proper specification becomes crucial not to affect understanding and exceed development effort and cost to a consumer application. This can be addressed through identifying antipatterns affecting services interfaces, and measuring their complexity by a set of metrics – e.g. the Baski & Misra metric suite. These two options are presented below.

2.1. Antipatterns

The catalogue of Antipatterns in [26] describes bad practices that make WSDL documents less readable. Antipatterns are concerned with how port-types, operations and messages are structured and specified in WSDLs. The reason behind adopting this antipattern catalog is three-fold. First, according to the recent survey in [27] of studies addressing Web Service antipatterns, this catalogue includes some of the antipatterns that are most frequently found in service-based systems. Second, this catalogue represents a good target since it includes antipatterns from several perspectives: problems related to high-level interface specification, syntactic and semantic issues (e.g., identifiers and comments), and message structure. Last but not least, to the best of our knowledge, this is the only antipattern catalog that has been deeply studied in the context of service-based systems derived from legacy code migration.

Table 1 summarizes the catalogue of antipatterns, classified in three categories: high-level service interface specification, comments and identifiers, and service message exchange. Consumer application developers prefer properly designed WSDL documents [26], so quality should be attended by service providers when building SOA frontiers. In the context of legacy to SOA migration scenarios, applying indirect migration favors achieving good WSDL document quality [30]. However, software engineers usually choose between direct or indirect migration based on classical criteria (e.g., time), disregarding those that may impact on readability and discoverability of SOA frontiers.

2.2. Baski and Misra's Metric Suite

The metric suite ("BM suite") presented in [4] is concerned with the effort required to understand data flowing to/from a service interface that can be characterized by the structures of messages used for data exchange. The BM suite includes four metrics, whose formulas are shown in Table 2, which can be computed from a service interface in WSDL. Metrics are explained as follows.

Table 1
Catalogue of WSDL antipatterns

Categories	Antipatterns	Symptoms
High-level service interface specification	Enclosed data model	Type definitions are placed in the WSDL rather than in separate XSD documents.
	Redundant port-types	Several port-types offer the same set of operations, on different binding types (e.g., HTTP, HTTPS, or SOAP).
	Redundant data models	Many types to represent the same domain objects within a WSDL.
	Low cohesive port-type	Port-types have operations with weak semantic cohesion.
Comments and identifiers	Inappropriate or lacking comments	(1) a WSDL document has no comments, or (2) comments are non explanatory.
	Ambiguous names	Ambiguous names are used for denoting the main elements of a WSDL document.
Service message exchange	Whatever types	A data-type might represent any domain object.
	Undercover fault information	Output messages are used to notify service errors.

Table 2
BM Metric Suite

Metrics Formulas
$DW(wsdl)=\sum_{i=1}^{n_m} C(m_i)$ (1) n_m: number of messages (input/output) of a WSDL $C(m)=\sum_{j=1}^{\#parts(m)} wp_j$ wp_j: weight value of the *j-th* part of message m
$DMR(wsdl)=\frac{DMC(wsdl)}{n_m}$ (2) n_m: total number of messages in the WSDL
$ME(wsdl)=\sum_{i=1}^{DMC(wsdl)} P(m_i)*(-\log_2 P(m_i))$ $P(m_i)=\frac{nom_i}{n_m}$ (3) nom_i: number of occurrences of the *i-th* message, n_m: total number of messages in the WSDL
$MRS(wsdl)=\sum_{i=1}^{DMC(wsdl)} \frac{nom_i}{n_m}$ (4) nom_i: number of occurrences of the *i-th* message, n_m: total number of messages in the WSDL

Data Weight metric (DW). This metric, with Formula (1), computes the structural complexity of service messages data-types. In (1), $C(m_i)$ broadly counts and weights the various XSD elements (simple/complex data-types) exchanged by the parts of message m_i. Each element or data-type definition in XSD is assigned with a weight value wp_j as a complexity degree. This wp_j is equals to w_e if the part references an element declaration in XSD, or w_t if the part references an XSD element definition. In turn, w_t depends on the data-type (simple or complex). See [4] for details about w_e and w_t. Then, DW values are positive integers. The bigger the DW of a WSDL is, the more dense the parts of its messages are. Then, DW values should be kept low.

Distinct Message Ratio metric (DMR). This metric, with Formula (2), considers that a WSDL may have many messages with the same structure. As the number of similarly-structured messages increases, less effort is likely needed to reason about them. Repetitive messages might allow to gain familiarity when inspecting the WSDL. The DMC function counts the number of distinct-structured messages in a WSDL, from the $\left[C(m_i), \#parts(m_i)\right]$ pairs, i.e., the complexity value and the total number of parts (input/output operation arguments) that each message contains. Then, the DMR metric is in range [0, 1], where 0 means that all messages are similarly-structured (lowest complexity), and 1 means that all messages are dissimilar (highest complexity). Then, DMR values should be kept low.

Message Entropy metric (ME). This metric, with Formula (3), exploits the percentage of similarly-structured messages that occur within a given WSDL. ME also assumes that repetition of the same messages makes a developer more familiar with the WSDL, but ME bases on an alternative differentiation among WSDLs in this respect. The ME metric has values in the range [0, $\log_2(n_m)$]. A low ME value means that messages are consistent in structure, i.e., the complexity of a WSDL is lower than others with equal DMR values [4]. Then, ME values should be kept low.

Message Repetition Scale metric (MRS). This metric, with Formula (4), analyzes variety in structures of a WSDL. MRS measures the consistency of messages by considering $\left[C(m_i), \#parts(m_i)\right]$ pairs in the given WSDL. MRS values are in the range [0, n_m]. A higher MRS means less effort to reason about messages structures due to repetition of similarly-structured messages. Then, MRS values should be kept high.

3. Related Work

Legacy system modernization to SOA has notably attracted research interest in the last decade. The most representative approaches for migration to SOA (according to our goals) are described as follows [29, 34].

3.1. Direct Migration

In [38], an approach is outlined to cut out selected pieces of legacy code and to provide them with an XML interface, by wrapping the navigation modules of a legacy system so that it can be accessed from a standard Web browser. This XML interface is used to generate a Java class, which acts as a proxy and creates XML messages returning from the server. The pieces of code are

selected, wrapped and reused as Web Services by employing a seven-step process: Function Mining, Function Wrapping, XSD Schema Creation, StubaServer Generation, Client Class Generation, Server Linking, and Web Service Binding. Individual pieces of legacy code are extracted by using a tool called SoftWrap, developed to automate the transformation of legacy program data-types into XML data elements. This tool supports languages such as PL/I, COBOL, and C/C++. Although this is highly automated, *a prior* study must be done to choose the legacy code to be migrated.

In [36, 37], legacy codes are wrapped with an XML shell which allows individual functions in the legacy programs to be offered as Web Services. Relevant pieces of functionality within the programs are identified by applying reverse engineering. For each piece of functionality to be wrapped, a new program (a subroutine of the parent program) is built, which is associated to a WSDL document. The author refers to this program as a subroutine with a call interface [37]. A SOA library is employed to package the new wrapped functionality. Finally, a proxy (as in the Proxy object-oriented pattern) is generated to link the Web Services to the underlying legacy business logic. Since created Web Services are stateful, the main drawback of the proposal is reentrancy. The state of the data contained within a wrapped Web Service is that of the last caller. Thus, if different clients are using the same service, their data might be mixed up. Besides, the work is suitable for small programs since the identification and exposition of the business functionality can be time consuming for bigger programs. Finally, the work focuses on migration costs and risks but Web Service interface quality is overlooked.

In [5], Canfora et al. proposed a black-box approach based on wrapping interactive legacy functionality and made it accessible as Web Services by using a Finite State Automaton, which describes the model of the interaction between users and the legacy system. The problem of transforming the original GUI of the system into the request/response scheme of a SOA system is solved by introducing a wrapper that is able to interact with the system on behalf of the user. In a subsequent work [6], the same authors illustrated how wrapping is used as part of a complete migration process consisting of the selection of services, wrapping the legacy functionality, and validation of the wrapped functionality. A main drawback of the work is that most of the work is done manually [1]. Another drawback is that the feasibility of state identification depends on the complexity interaction patterns within the legacy GUI.

Zhang et al. [41] proposed a black-box approach to export customized interactive functionality in legacy systems as Web Services using a wrapping technique suitable to GUI-based legacy systems. The work also proposes a solution to deploy such Web Services that consist of a mediation support between users and legacy systems in a SOA deployment. A distributed framework that executes Web Services and integrates graphical user interfaces of legacy systems is also presented. GUI commands are wrapped as Web Services placed in a service container. Each Web Service consists of a Python script and a WSDL document. The authors presented a case study to show the feasibility of the proposal, where two GUI-based legacy systems – Rational Rose and Computer Associates Erwin – were migrated to SOA. A drawback of this work is that legacy business (logic) code is not considered to be exposed as Web Services.

The work of Millard et al. [24] presents three design patterns for wrapping legacy systems as Web Services, and suggests implementation guidelines, applicability and certain consequences of the patterns. The Lowest Common Denominator Interface, Most Popular Interface and the Negotiated Interface patterns are used to create a common interface for two or more software components that share some common functionality. Similar software components should be wrapped with a common interface to enable them to be used modularly within the resulting SOA system. The authors derived these patterns from two Item Bank systems. Item Banks are databases storing questions that can be queried to provide content for either summative or formative assessment. Item Banks have slightly different functionality (i.e., only queries differ) and use different data formats to store their questions [24]. This work also deals with candidate services identification and grouping similar operations into Web Services. However, the authors do not include quantitative or qualitative analysis about the systems obtained by applying these patterns.

3.2. Indirect Migration

Chung et al. [10] described a project in which a legacy tool called Bertie3 was reengineered to SOA resulting

in a new tool, Service-Oriented Bertie (SoBertie). The core of Bertie3 was hosted on a server and exposed as a Web Service. Besides, Web Service clients were created to consume the services by basing on the required functionality. In a follow-up work [9], the authors presented a reengineering methodology called Service-Oriented Software Reengineering (SoSR) designed for migrating legacy systems to SOA. SoSR is conceptualized from a three service participants model, a 4+1 view [19] and responsibility assignment charts. SoSR can be used by software engineers to modernize highly coupled legacy systems, generating new decoupled, agile and service-oriented systems. However, the methodology might be complex due to the need of using several views. In addition, SoSR mixes different software architectures: 3-tier for business logic design, n-tier for service deployment, and SOA for integrating the legacy system with the new environment.

Distante et al. [13] presented a generic framework to re-design legacy systems for the Web, using UWA [18] and its extended version called UWAT+ [12]. The UWA design framework offers the designer metamodels and tools for user-centered design of data and operation-intensive Web applications. Based on this, the work in [13] blends design recovery technologies for capturing the know-how embedded in a legacy application with forward design methods suited for Web-based systems. The work consists of designing technologies for recovering legacy information, by following a three-step process: Requirement elicitation, Reverse engineering and Forward design. This work has been evaluated by migrating a legacy system to support hiring of personnel. The main disadvantage of the work is a lack of hints or guidelines on the applicable technologies for the above listed three steps, which forces practitioners to fill this gap.

In [7], an approach based on product feature analysis is proposed for migrating legacy systems to SOA. A "feature" is a coherent and identifiable bundle of system functionality that is visible to the user via a GUI [7]. Then, feature analysis addresses the understanding of features in software systems and defines mechanisms for carrying a feature from the problem domain into the solution domain. The approach involves detecting desired features and identifying the legacy modules implementing them, and relies on feature identification, model feature construction and feature implementation detection upon the legacy system. Some of the features considered involve semantic similarity between legacy functions and data granularity.

Cuadrado et al. [11] proposed a white-box approach for reengineering legacy code based following a three-step process, which involves legacy architecture recovering, evolution plan creation, and plan execution. As a clear benefit, the architecture recovering process includes incorporating documentation. From a technical perspective, the work uses QAR, a workflow for architecture recovery based on three main activities: documentation analysis, static analysis and dynamic analysis. As a basis for the new architecture the authors propose OGSi [39], a standard framework for service execution plus facilities for service lifecycle management. Finally, the evolution is completed by generating a set of services. According to the authors, this approach is suitable for medium-sized systems preferably.

SOAMIG [43] is a COBOL to SOA migration methodology and supporting tools. The migration methodology consists of four phases, each carried out in several iterations. The main idea behind SOAMIG is to transform the original system into a SOA system by using several translation tools. For instance, it proposes to use a tool that translates COBOL code to Java code, which is easier to expose as Web Services. However, this methodology might negatively impact the quality of the SOA frontier because COBOL code was designed using out-of-date design criteria, and this kind of tools do not actually redesign the old system. In addition, COBOL imposes some length limitations to routine names and comments that might be translated into a SOA frontier, which in turn might represent a quality issue for the frontier.

3.3. Combining Direct and Indirect Migration

Architecture Reconstruction and MINing (ARMIN) [25] is a proposal to identify and use legacy components as services. ARMIN uses the information extracted from legacy code to identify candidate services based on the dependencies between legacy components. Dependencies include functional dependencies, where a component uses functionalities of other system components, and data dependencies, where global data are shared by several system components. Then, ARMIN uses the data to generate a

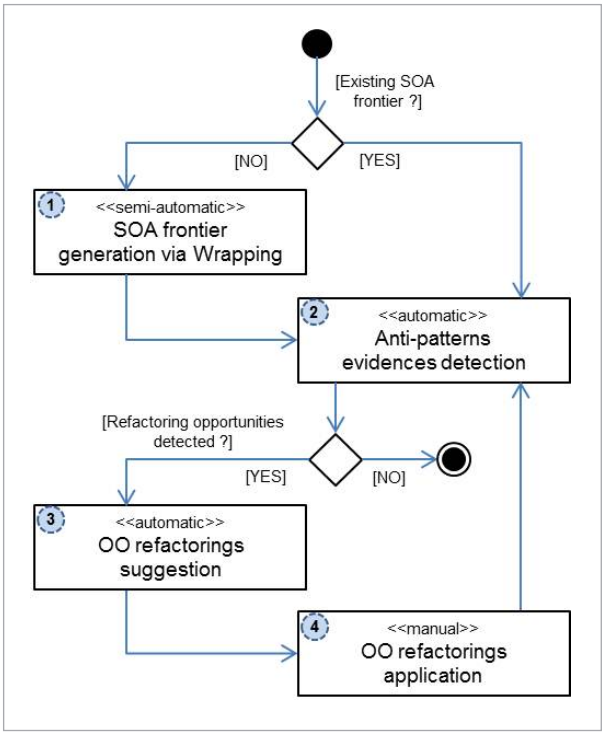
component view. The authors applied ARMIN on a legacy system written in C++, comprising 800,000 lines of code and 2,500 classes. The authors reported issues due to implementation and documentation inconsistencies, indicating the need for deeply studying the code prior to migration.

In [20, 21, 35], the Service-Oriented and Reuse Technique (SMART) is discussed. SMART aims at helping organizations to decide whether their legacy functionality can be exposed as Web Services. SMART has been designed to modernize military systems by applying wrapping, but it has evolved into a new version, taking into account those cases where wrapping is not an option. SMART considers specific interactions prescribed by SOA, and considers any modification to be performed upon the legacy modules. With SMART, a wide range of information about legacy components, the target SOA, and potential services to produce a service migration strategy is gathered from meetings with stakeholders. This activity is directed by the Service Migration Interview Guide (SMIG), a set of questions that address the gap between the existing and the target architecture, design, and source code, and questions concerning issues that must be addressed in service migration efforts [21]. SMART starts with an architectural and design step, followed by a gap analysis between the target SOA system and the original system. In particular, the gap analysis suggests trade-offs between the original and the target architectures. Finally, a migration technique is selected (e.g., quick and dirty [21] or wrapping). Interestingly, SMART is flexible since it allows engineers to combine several migration approaches, but it does not provide tools for assisting developers in executing the migration.

4. COB2SOA Migration Approach

COB2SOA relies on refactoring opportunities applied upon the SOA frontier of a legacy system, by detecting evidences of antipatterns [31]. The hypothesis is that enhancing the SOA frontier of a wrapped legacy system can be done in a cheap and fast way by analyzing legacy source code and WSDL interfaces, and supplying developers with guidelines (refactoring opportunities) for manually refining WSDLs based on the evidence of antipatterns. The main activities of COB2SOA are shown in Figure 2.

Figure 2
COB2SOA: Main activities



The input is a legacy system source code (COBOL files) and the set of WSDL documents that result from a direct migration. If the legacy system does not have a SOA frontier yet, a semi-automatic process generates one Web Service per COBOL program. As a pre-processing, COBOL data-types are converted into XSD data-types for WSDLs. Data exchanged by COBOL programs are manually identified to then create a wrapper for it (e.g., using .NET or Java). Then, the WSDL documents are automatically generated from the source code of the wrapper. The initial SOA frontier is automatically analyzed to detect evidences of WSDL antipatterns. Then a list of concrete suggestions is generated to improve the services frontier. After that, developers apply all or some of the suggested refactoring actions. These steps can be done in successive iterations and refinements, where a new and improved SOA frontier can be obtained.

4.1. Heuristics on Antipatterns Evidences

To detect the antipatterns evidences (root causes), some heuristics were defined and implemented – shown in Table 3 – allowing assistance during the sys-

Table 3
Heuristics on Antipatterns evidences

Evidences	Input	Detected when
E1: Shared dependencies among two service implementations	COBOL	Given the list of COBOL programs that are copied, or included, or called from two or more service implementations; When the list is not empty.
E2: Data overlapping	WSDL	An XSD complex data-type subsumes another complex XSD data-type, or a list of parameters subsumes another list of parameters.
E3: Too many input/output parameters	WSDL	At least one operation input/output has more than parameters.
E4: Redundant data-types definitions	WSDL	At least two XSD data-types are syntactically and structurally identical.
E5: Inconsistent data-types	WSDL	The name of a parameter denotes a quantity but it is not associated with a numerical data-type.
E6: Unused parameters	COBOL	At least one parameter is not associated with a COBOL MOVE statement.
E7: Semantic similarity of services and operations	WSDL	The names (and their documentation) of two services or operations, are near in a vector space model [38].
E8: Lack of documentation	WSDL	At least one operation lacks comments in the documentation element.
E9: Inappropriate naming convention	WSDL	An operation name contains more than one verb, or a parameter name contains a verb, or a name token is less than 3 in length, or tokens refer to a specific technology.
E10:Error information being exchanged as output data	WSDL	An output message has tokens like: "*error*", "*fault*", "*fail*", "*exception*", "*overflow*", "*mistake*", "*misplay*», etc.

tem migration. Most of the heuristics help to analyze WSDL documents to detect evidences of antipatterns that may affect service readability and discoverability (see Section 2).

COB2SOA also focuses on evidences of bad practices in COBOL source code. Two antipatterns for COBOL source code were added, namely unused parameters and shared dependencies among two COBOL programs representing service implementations. To detect evidences of these two antipatterns, COBOL code files are reverse engineered with an analysis of the common area for data-type exchange, called COMMAREA[1].

Evidences of antipatterns can be combined to create refactoring opportunities. Six refactoring opportunities have been defined, referred as Legacy-System-to-SOA (LSSOA) refactorings according to the following combinations:

- $E1 \wedge E2 \rightarrow R1$: Remove redundant operations
- $E2 \rightarrow R2$: Expose shared programs as services
- $E3 \vee E4 \vee E5 \vee E6 \rightarrow R3$: Improve business object definitions
- $E7 \rightarrow R4$: Improve service operations cohesion
- $E8 \wedge E9 \rightarrow R5$: Improve names and comments
- $E10 \rightarrow R6$: Improve error handling definitions.

1 The COMMAREA option specifies the name of a data area in which data are passed to the program being invoked.

4.2. SOA Refactoring Opportunities

The six refactoring opportunities applicable over a SOA frontier are now briefly explained. They allow users to remove the evidences of the ten antipatterns explained in the previous section. Table 4 shows how LSSOA refactoring opportunities are related (into one or more logical combinations) to OO refactorings from the Fowler et al.'s catalogue [15]. The rationale of this is that conceptually services are described as OO interfaces exchanging messages, with data-types described using XSD. For specific details concerning refactoring opportunities, the reader is referred to [31].

R1: *Remove redundant operations.* This refactoring is similar to duplicate code in other contexts [26]. This

Table 4
SOA frontier refactorings and Fowler et al.'s refactorings

LSSOA refactorings	OO refactorings
R1: Remove redundant operations	1: Extract Method ⇒ Extract Class
R2: Expose shared programs as services	1: Extract Method ⇒ Extract Class
R3: Improve business object definitions	1: Convert Procedural Design to Object ; Replace Conditional with Polymorphism 2: Inline Class 3: Extract Class ; Extract Subclass ; Extract Superclass ; Collapse Hierarchy 4: Remove Control Flag ; Remove Parameter 5: Replace Type Code with Class ; Replace Type Code with Subclasses
R4: Improve service operations cohesion	1: Inline Class ; Rename Method 2: Move Method ⇒ Move Class
R5: Improve names and comments	1: Rename Method ⇒ Preserve Whole Object ⇒ Introduce Parameter Object ⇒ Replace Parameter with Explicit Methods
R6: Improve error handling definitions	1: Replace Error Code With Exception

Numbers means the steps to be performed to fulfill the OO refactorings.
"⇒" means that only one OO refactoring should applied.
";" means that all OO refactorings should be applied in that strict order.

can be detected both in COBOL source code and in WSDLs. Each service exposes an interface that wraps COBOL programs, whose business logic involves an interface including COBOL data-types and dependencies to other programs. The Extract Method OO refactoring may be applied, to create a single operation in a WSDL (grouping several redundant operations) to be invoked for all the points where the redundancy was detected. At the class level, the Extract Class is applied to generate a new service from the redundant services.

R2: *Expose shared programs as services*. Usually some programs/routines contain functionality that represents core business itself. Such routines might have several client routines dependents, representing highly reusable business logic modules. Exposing these routines as services can reduce the chance of redundant operations, increasing the possibility of Web Services composition [2, 17]. The Extract Method OO refactoring can be applied, which is similar to having a long method. In SOA, it means generating new service operations that might help service consumers to identify the requested functionality. If a decomposition of a long routine exposes several service operations, a new service could be generated, i.e., by applying Extract Class.

R3: *Improve business object definitions*. This can be detected with the lack of a single XSD file for a set of services within the same frontier. This means, a bad business model definition (or the lack of a unique data-type schema) hinders the general readability of services and their reusability. As shown in Table 3, up to five steps and many OO refactorings should be done here. In fact, several aspects of a service are involved: Too many output/input parameters, Redundant data-types definitions, Data-types with inconsistent names and types, and Unused parameters. Some of them can be solved with one OO refactoring, while others require more than one in alternate combinations. For example, Too many output/input parameters, evidences the use of many variables as parameters of procedural modules, that should be arranged among different business objects. The Convert Procedural Design to Object OO refactoring can be applied, to restructure common data-types schema for a set of Web Services.

R4: *Improve service operations cohesion*. This improvement consists of grouping semantically similar operations and/or Web Services in terms of business functionality. For similarity between two services, a 1-to-1 association is assumed between COBOL pro-

grams and Web Services, i.e., produced Web Services contain a single operation. Therefore, this also represents the similarity between two COBOL programs. This implies some aspects with different alternatives as solutions. For example, the Move Method OO refactoring can be applied, which is used to re-locate methods being odd within a class, and mostly invoked by other classes. In SOA, this would be equivalent to moving operations between services. When a group of similar services (of one operation) is identified, a new service could be built by applying Move Class.

R5: *Improve names and comments*. WSDLs must precisely describe how to invoke certain functionality as well as the meaning of that functionality. This implies to improve names of operations, messages, porttypes, parts and elements present in a WSDL, by adding documentation according to the meaning of the service. Improving these elements implies dealing with semantics, and hence they cannot be fully automated. This situation implies several alternatives. For example, renaming service operations is equivalent to apply the Rename Method OO refactoring.

R6: *Improve error handling definitions*. This is related to improper handling of errors and exceptions in services, which occurs when WSDL operations exclude <fault> elements. Instead, errors are exchanged along with pure data. Thus, the actual result (correct or error) from an operation is unknown until invoking the service. The Re-place Error Code With Exception OO refactoring can be applied here. This refactoring adds the missing <fault> elements to WSDLs and the refinement of data-types mixing output and error data. A textual description (string) or a <complexType> can be used to report details of the error.

5. Experimental Evaluation

This section evaluates the effectiveness of migration approaches to SOA regarding frontier complexity in terms of the BM suite (see Section 2). Two case studies, concerning COBOL systems, have been subject of migration to SOA by employing the three approaches: Direct and Indirect migration, and COB2SOA. The first case study is the legacy system of the largest Argentinian government agency [30]. The second case study is a legacy system providing support for mill sales management [31].

5.1. Direct and Indirect Migration

The first case study involves a 35-year-old system manages information of the entire population in Argentina [30]. It runs on an IBM AS/400 mainframe. Some programs are used via an intranet and others are grouped in CICS transactions that are consumed by Web applications. Direct migration was done by wrapping the CICS-enhanced programs, creating a preliminary Web Services frontier. One 1-operation Web Service for each COBOL program was generated, adding a thin C# .NET service layer. Then, WSDL files were automatically generated from the C# source code. Developers migrated 32 COBOL programs, generating 32 services in about 5 days. Indirect migration was done to re-implement the 32 COBOL programs in C#. An indirect SOA frontier was built, consisting of 7 services, and 1 XSD file representing a single data model. The generated WSDL files were manually refined until an antipatterns-free SOA frontier was obtained. The whole migration process demanded 13 months: 1 month to manually define WSDL files, 3 months to analyze legacy functions, 1 month to refine WSDL files, 6 months to rewrite the business logic, and 2 months to test the obtained indirect SOA frontier.

The Mill Sales Management system provides support for sales transaction management between clients, suppliers and creditors [31]. The system comprises 211 COBOL programs and an extra COBOL program acting as a program selector (menu). No databases or CICS transactions are involved. Data storage is programmatically handled via ".dat" files. An independent file – the COMMAREA – is used for data definition of each COBOL program. First, direct migration was done, generating 211 Web Services by wrapping each COBOL program via a 1-to-1 mapping strategy. WSDL interfaces were also generated by building a thin service layer. The migration to SOA demanded 21 days. Then, indirect migration of the original system was also done. Similar to the first case study, the goal was to generate a high quality SOA frontier, i.e., without antipatterns. After 6 months, 50 Web Services were built – 2 months to analyze the legacy functions, 1 month to design the WSDL files, and 3 months to refine the WSDL files. Since this case study had experimental purposes only, no actual deployment was done. Hence, testing the obtained indirect SOA frontier was left out – which might require more than 2 months compared to case study 1.

5.2. COB2SOA Migration

The direct SOA frontiers of both case studies were taken as input for the COB2SOA approach, to identify refactoring opportunities (see Section 4.2). Table 5 summarizes the refactoring opportunities detected. After that, the new SOA frontier – the set of refactored WSDL documents – for both case studies were obtained.

For case study 1, there were refactoring opportunities applicable to all the services/programs and others applicable to a subset of the services/programs. Besides, a suggestion came up that the 32 services should be grouped in just 16 services. One specialist applied the proposed refactorings in 2 days, to create the new SOA frontier of 16 services + 1 XSD file. For case study 2, one refactoring opportunity was applicable to all the services/programs (211) and other was applicable to most of them (209 services). Finally, after manually applying the proposed refactorings (in about 1 month), the new SOA frontier was created containing 115 services + 1 XSD file.

5.3. A Comparison of Service Interfaces Quality

SOA frontiers obtained by direct migration strongly depend on the original system design. In turn, interfaces obtained by indirect migration might be more independent [30], since the legacy system functionality is re-implemented using modern technologies and new design criteria. The main goal of this work is to assess the trade-off between cost/time and services frontier quality. Thereby, setting forth empirical evidence can reveal how a migration approach influences a SOA frontier quality – mainly focusing on the complexity level according to the BM suite (see Section 2). In addition, a quantitative analysis highlights some results from the migration processes, as follows.

Quantitative Analysis. The first advantage of indirect migration and COB2SOA compared to direct migration is the unique XSD document generated, to share the definition of common data-types across all WSDL files. In addition, the fewer number of WSDL files means they include more operations, fostering a functional definition of related cohesive operations within a WSDL.

Table 6 shows the quantitative results of case study 1. The number of offered operations across alternatives was 38, 45 and 43 for direct migration, indirect migration and COB2SOA migration, respectively. While 32 COBOL programs were originally migrated, direct

Table 5
LSSOA refactorings identified on both case studies

LSSOA refactoring	Case study 1 (32 services)	Case study 2 (211 services)
R1: Remove redundant operations	7 redundant operations detected	35 redundant operations detected
R2: Expose shared programs as services	6 COBOL programs to be exposed	0 COBOL programs to be exposed
R3: Improve business object definitions	32 services needed to be improved	209 services needed to be improved
R4: Improve service operations cohesion	16 services identified	115 services identified
R5: Improve names and comments	32 services needed to be improved	211 services needed to be improved
R6: Improve error handling definitions	32 services needed to be improved	0 services needed to be improved

Table 6
Case study 1: General quantitative results

Migration attempt	WSDLs	Operations	Time	LOC per file	LOC per operation	Comments per file
Direct Migration	32	38	5 days	157.25	129	0.00
Indirect Migration	7 + 1 XSD	45	13 months	495.50	88	30.25
COB2SOA	16 + XSD	41	2 days	235.35	97	16.00

migration resulted in 38 operations because one program was divided into 7 operations. After manually analyzing the business logic, the expert staff determined that only 2 of those operations were useful and the remaining operations were marked as duplicate. In this context, indirect migration and COB2SOA migration resulted in more operations. There are two main reasons for this: disaggregating functionality and exposing shared functionality. Disaggregating functionality means that certain services/programs, which returned more than 100 output parameters, had various purposes and were mapped to several purpose-specific operations. The second reason is that several COBOL programs shared dependencies, i.e., other COBOL programs not yet exposed as services.

The number of LOC per operation for indirect migration was the lowest. Interestingly, COB2SOA migration resulted in a slightly higher number of LOC per operation than indirect migration. In contrast, the number of LOC per operation resulting from applying direct migration was more than twice the number of LOC obtained through the other two migration approaches. Regarding LOC per file –157.25, 495.50 and 235.35 for direct, indirect and COB2SOA migrations, respectively.

Figure 3 illustrates the differences between the three approaches. Interestingly, for indirect migration and COB2SOA migration the number of documents decreased in comparison with direct migration. This means that, after applying direct migration, a service consumer must read more WSDL code to understand how to invoke an operation. Regarding COB2SOA, the LOC is similar to the LOC achieved in indirect migration, but involving more documents. This means that COB2SOA ended up with a different operation layout across services, leading to smaller average LOC per file.

Figure 3
Case study 1: Total LOC in WSDL/XSD documents

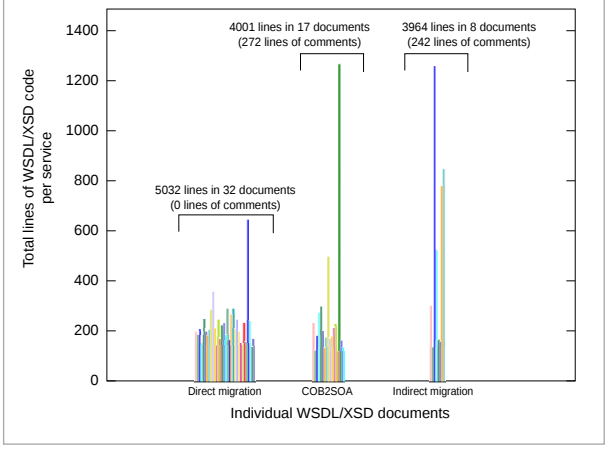


Table 6 also shows the number of comments per WSDL/XSD document. The WSDL documents from direct migration did not include comments because the tools for generating WSDLs documents in case study 1 are unable to read COBOL comments and move them to the WSDL documents through COMTI wrappers. In addition, developers did not place effort in including comments manually because of the tight schedule, which a typical situation in practice when following direct migration [31]. Figure 3 depicts the number of comments for each migration. Direct migration resulted in a total of 5,032 lines, with no comments. In contrast, although the total LOC in indirect migration and COB2SOA migration was lower, 242 (6.10%) and 272 (6.79%) lines were comments, respectively.

Table 7 shows the quantitative results of case study 2. Direct migration generated 211 documents because of the 1-to-1 mapping of COBOL programs for generating the WSDL documents. In contrast, indirect migration and COB2SOA migration generated much less documents. Then, the number of offered operations was 252, 202 and 206 for direct migration, indirect migration and COB2SOA migration, respectively. In this case, several COBOL programs were used for storing clients, suppliers, creditors, payments, and products. In general, these programs fell into one of three types of transactions: add, update and delete. These programs were migrated by providing related operations according to their type (e.g., add, update, or delete data). That is the reason why there were 252 operations in 211 services. In turn, indirect migration and COB2SOA migration resulted in 202 and 206 operations, respectively. There are fewer operations due to redundant operations, which were removed in both cases.

Regarding the LOC per operation, direct migration, indirect migration and COB2SOA migration resulted in 154, 60 and 80 units, respectively. As depicted in Table 7, the number of LOC per operation for indirect migration operation was the lowest. Furthermore,

Table 7
Case study 2: General quantitative results

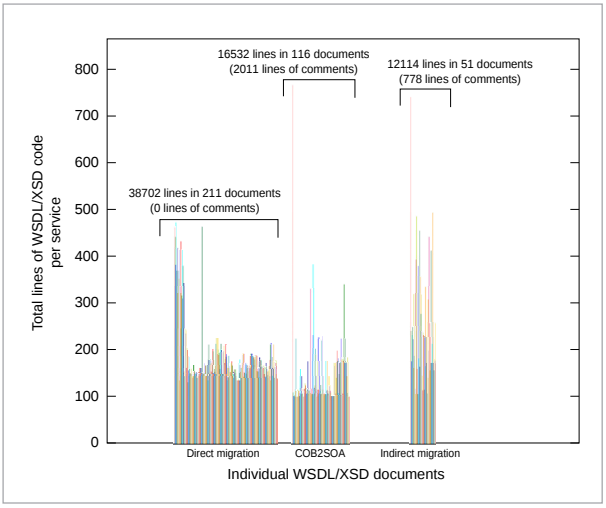
Migration attempt	WSDLs	Operations	Time	LOC per file	LOC per operation	Comments per file
Direct Migration	211	252	21 days	183.42	153	0.00
Indirect Migration	50 + 1 XSD	202	6 months	237.52	60	15.25
COB2SOA	115 + 1 XSD	206	1 month	142.52	80	17.33

similar to case study 1, the COB2SOA migration resulted in a higher number of LOC per operation than indirect migration, but lower than direct migration LOC, which was 154.

Moreover, Figure 4 illustrates the total LOC for the three frontiers. Applying indirect migration resulted in a total of 12,114 LOC in 51 WSDL/XSD documents. Similarly, the COB2SOA migration produced a total of 16,532 LOC in 116 documents. Finally, direct migration contained a total 38,702 LOC in 211 documents. In this case, it is also clear that the total LOC for indirect and COB2SOA migrations was smaller than that of direct migration. In this case study, the number of documents for indirect and COB2SOA migrations were smaller in comparison with direct migration.

Finally, the WSDL documents resulting from direct migration for case study 2 did not contain comments. The idea was to simulate a real-life situation, where service developers in general do not write comments

Figure 4
Case study 2: Total LOC in WSDL/XSD documents



or the comments are not clear enough to understand the functionality exposed by the services. In contrast, as indirect migration and COB2SOA migration are intended for generating high-quality WSDL documents, documentation was added to the SOA frontiers in both cases. Related to this fact, Figure 4 shows the number of comments for each migration alternative related to the total LOC for each frontier. As it can be seen, direct migration had a total of 38,702 lines, with no comments. In contrast, although the total LOC in indirect migration and COB2SOA migration was lower (similar to case study 1), 778 (6.55%) and 2011 (12.27%) lines were documentation, respectively.

Antipatterns Assessment. Table 8 summarizes the antipatterns detected when applying direct migration, indirect migration and COB2SOA migration on both case studies. A manual review by a specialist on WSDL documents of the three migration approaches was made. The results show that the resulting WSDL files of direct migration have more antipatterns than documents of COB2SOA migration, while there were no antipatterns in the WSDL files of indirect migration.

The first row describes an antipattern that is generated by many code-first software, which force data models to be included within the generated WSDL documents. In contrast, neither indirect migration nor COB2SOA migration were affected by this antipattern. Similarly, the second row describes an antipattern that ties abstract service interfaces to specific communication protocols or implementations, hindering black-box reuse [32]. In general, this is caused by the use of defective tools to translate source code to WSDL code. To avoid this antipattern in C# (case study 1 language), developers should supply C# service codes with special annotations, so they are processed by these tools.

The antipattern described in the third row is related to poor data model designs. Redundant data models

usually arise from limitations or bad use of the software to generate WSDL documents. This antipattern only affected WSDL documents generated through direct migration. Although there were no repeated data-types at the WSDL documents level, COB2SOA migration produced repeated data-types at a global level, i.e., when the data-types in all documents are taken into account. For example, in case study 1 the "error" data-type, which consisted of a fault code, actor and description is repeated in all the WSDL documents of COB2SOA migration. This is because this data-type has been derived several times from different sub-systems. Finally, this did not happen when using indirect migration because the WSDL document designers had a global view of the system.

The fourth antipattern means having no semantically related operations within a port-type. This antipattern did not affect WSDL documents generated through direct migration or indirect migration. Direct migration documents were not affected because almost all WSDL documents included only one operation, while indirect migration WSDL documents were specifically designed to group related operations. However, COB2SOA migration uses an automated process to select which operations should be grouped into a specific port-type. In the experiments, when several related operations in a service used the same unrelated programs such as text-formatting programs, the COB2SOA migration suggested that these routines were also a candidate operation for that service. This results in services that had port-types with several related operations, but few unrelated operations.

The fifth and sixth rows describe antipatterns that impact on the comments and names in services [32]. Names in the resulting WSDL documents were too short and difficult to be read. The reason is that names in COBOL programs have length restrictions and they were directly mapped to WSDL documents. This also caused lack of documentation in WSDL documents, since WSDL generation tools operate on service binary codes and hence in general disregard service code comments [33]. On the other hand, ambiguous names affect WSDL documents in COB2SOA migration only when the original COBOL program was designed using control couples. This is because properly putting representative names and documenting this kind of couples is known to be a complex task [40].

The antipattern in the last row of the table deals with errors being transferred as part of output messages, which for direct migration of case study 1 resulted from the original transactions that used the same COMMAREA for returning both output and error information. In contrast, the WSDL documents of indirect and COB2SOA migration had an adequate

Table 8
Antipatterns detected in WSDL documents of both case studies

Antipatterns	Direct Migration	Indirect Migration	COB2SOA
Enclosed data model	Always	Never	Never
Redundant port-types	Several communication protocols are supported	Never	Never
Redundant data models	Two operations use the same data-types	Never	Never
Low cohesive port-type	Never	Never	Several related programs use an unrelated operation, such as text formatting routines
Inappropriate or lacking comments	Always	Never	Never
Ambiguous names	Always	Never	Never
Undercover fault information	Data-types with names indicating error messages exchange business logic data	Never	Never

mechanism for error handling based on standard fault messages provided by WSDL. For the second case study, direct WSDL documents did not present the antipattern, because no data-type was originally defined for exchanging error information. On one hand, the direct SOA frontier for both case studies did not contain fault messages in the WSDL documents. On the other hand, only the direct frontier of the first case study included undercover error information exchanged with the business logic data. For case study 2, COMMAREAs were embedded in COBOL programs. By reviewing the DATA DIVISION section of COBOL programs, where all the variables to be used in the business logic must be declared, it could be seen that the business data were completely separated from the error data structure definitions.

Complexity Measurement. The BM metric suite was applied on both case studies upon the three migration approaches. The aim is to evaluate how a migration approach influences the complexity level of a generated SOA frontier – with a likely impact on comprehension and interoperability. Before analyzing measurement results from both case studies, we recall the expected (good/bad) values on each metric of the BM suite.

Figure 5 shows the results for case study 1. The COB-2SOA migration outperforms the direct migration in both DMR and MRS. However, the indirect migration obtained the best values for these two metrics – i.e., the lowest DMR and the highest MRS. This means that refactorings have produced higher use of similarly-structured messages, and this improves WSDL comprehension. However, the ME metric was affected by COB2SOA and indirect migration – the worst value (highest) for indirect migration. This means, a high distribution of similarly-structured messages, instead of being consistent in structure. Finally, the DW metric was also affected by COB2SOA and indirect migration – obtaining higher values than for direct migration. This means, refactorings have produced a larger number of complex data types – probably with a goal to better reflect business domain objects (see Table 4).

Results for case study 2 are shown in Figure 6. The general trends are quite similar to case study 1 for the DMR, MRS and DW metrics. However, the ME metric was particularly benefited by COB2SOA, obtaining the lowest (better) value. Then, after the refactorings, most WSDL files resulted with few data structures largely repeated (w.r.t. other similar-structures) being as such highly consistent within the given WSDL files.

Discussion. From the quantitative and qualitative analysis above, it can be seen that the COB2SOA migration approach has a midway performance. Regarding the time vs. quality trade-off there is a better performance, considering a quality trend towards the indirect approach, but with a time trend very close to the direct approach and largely far from the indirect migration. As such, the COB2SOA approach comes up as an optimized option for the industry when engaging in a COBOL to SOA migration.

Figure 5
Case Study 1: BM suite metrics upon migration approaches

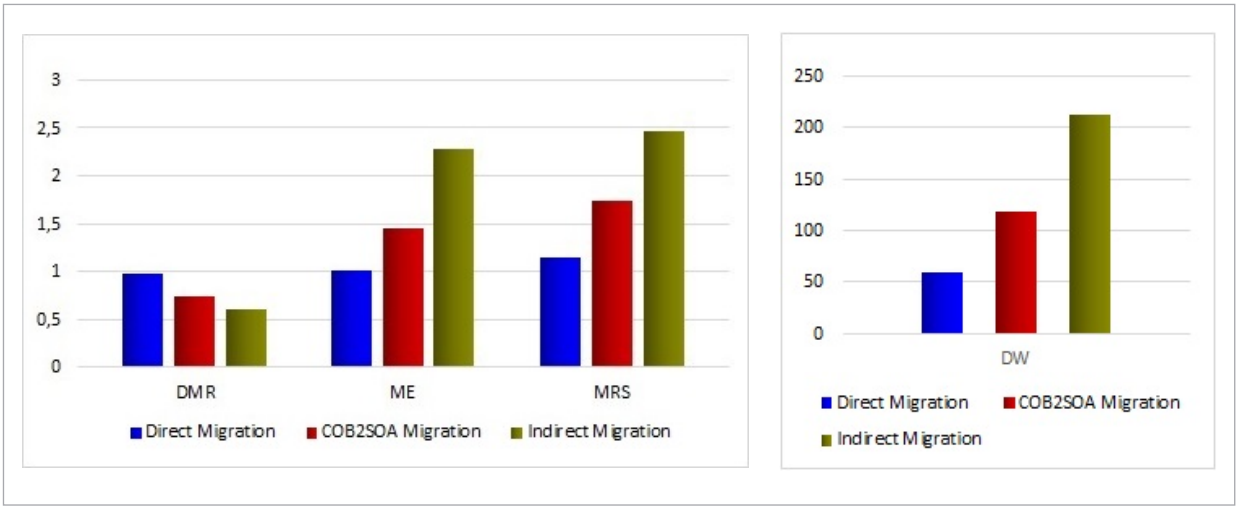
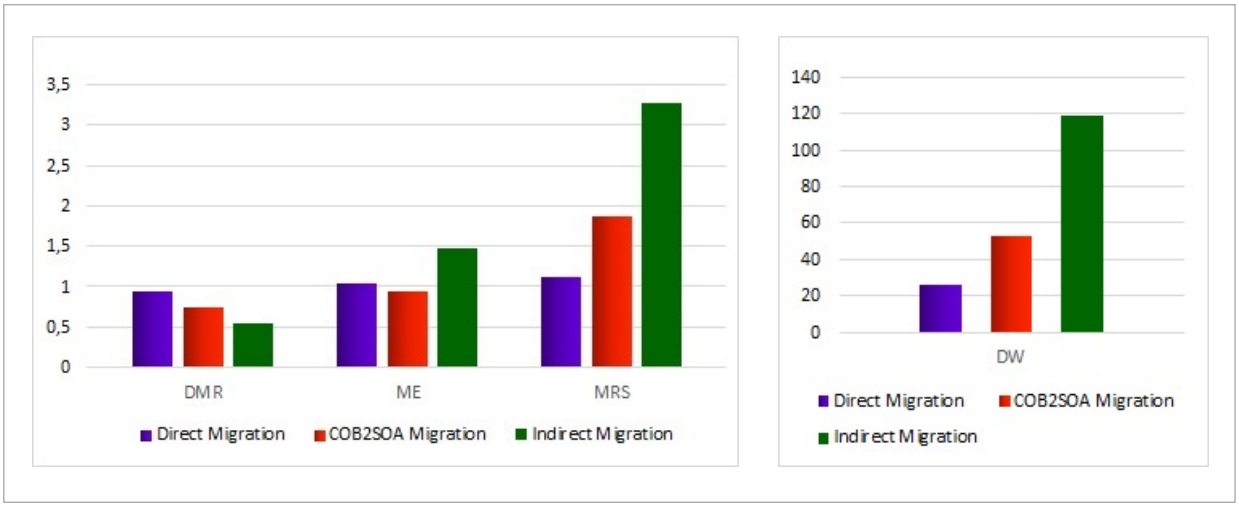


Figure 6
Case study 2: BM suite metrics upon migration approaches



6. Conclusions and Future Work

This paper studied how direct and indirect COBOL to SOA migration approaches perform regarding the quality of the generated SOA frontier. In addition, an intermediate approach, called COB2SOA, is also evaluated. Qualities attended by COB2SOA and the indirect migration are related to readability and discoverability of the produced WSDL documents. In particular, a set of antipatterns – bad practices – is considered at the levels of WSDL and COBOL source code. In this study, we also evaluated the complexity level of a SOA frontier, by using the BM metric suite. Complex WSDL documents may impact on comprehension and interoperability, which might affect new business relationships of a target organization. After a quantitative and qualitative analysis, a conclusive evidence arises in favor of the COB2SOA approach, in terms of balanced trade-off between quality and time.

As future work, we expect to conduct another study concerning well-known OO metrics from Chidamber & Kemerer [8], from which we have found a correlation with the BM metric suite [3, 23]. By considering the OO back-ends of the SOA frontiers that are generated in the three approaches we might early analyze another refactoring opportunities to then generate improved WSDLs with lowest complexity – i.e., increasing comprehension and interoperability.

Acknowledgement

This work is supported by the ANPCyT grant no. PICT-2013-0464, the ANPCyT grant no. PICT-2017-1725, the CONICET grant no. PIP 2017-2019 GI 11220170100951CO, and the SPU-UNCo grant no. PIN I 2017-2020 04-F009.

References

1. Almonaies, A., Cordy, J., Dean, T. Legacy System Evolution Towards Service-Oriented Architecture. Proceedings of the International Workshop on SOA Migration and Evolution (SOAME), Madrid, Spain, March 2010, 53-62.
2. Alrifai, M., Skoutas, D., Risse, T. Selecting Skyline Services for QoS-Based Web Service Composition. Proceedings of the 19th ACM International Conference on World Wide Web (WWW), Raleigh, North Carolina, USA, April 26-30, 2010, 11-20. https://doi.org/10.1145/1772690.1772693
3. Anabalon, D., Flores, A., Mateos, C., Zunino, A., Misra, S. Controlling Complexity of Web Services Interfaces through a Metrics-driven Approach. Proceedings of the IEEE International Conference on Computing Networking and

Informatics (ICCNI), Lagos, Nigeria, 29-31 October, 2017, 1-9. https://doi.org/10.1109/ICCNI.2017.8123807

4. Baski, D., Misra, S. Metrics Suite for Maintainability of eXtensible Markup Language Web Services. IET Software, 2011, 5(3), 320-341. https://doi.org/10.1049/iet-sen.2010.0089
5. Canfora, G., Fasolino, A., Frattolillo, G., Tramontana, P. Migrating Interactive Legacy Systems to Web Services. Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR), Bari, Italy, 22-24 March, 2006, 24-36. https://doi.org/10.1109/CSMR.2006.34
6. Canfora, G., Fasolino, A., Frattolillo, G., Tramontana, P. A Wrapping Approach for Migrating Legacy System Interactive Functionalities to Service Oriented Architectures. Journal of Systems and Software, 2008, 81(4), 463-480. https://doi.org/10.1016/j.jss.2007.06.006
7. Chen, F., Li, S., Yang, H., Wang, C.-H., Chu, C.-C. Feature Analysis for Service-Oriented Reengineering. Proceedings of the 12th IEEE Asia-Pacific Software Engineering Conference (ASPEC), Taipei, Taiwan, 15-17 December, 2005, 201-208. https://doi.org/10.1109/APSEC.2005.67
8. Chidamber, S., Kemerer, C. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 1994, 20(6), 476-493. https://doi.org/10.1109/32.295895
9. Chung, S., An, J., Davalos, S. Service-Oriented Software Reengineering: SoSR. Proceedings of the 40th IEEE Annual Hawaii International Conference on System Sciences (HICSS), Waikoloa, HI, USA, 3-6 January, 2007, 172c–. https://doi.org/10.1109/HICSS.2007.479
10. Chung, S., Young, P., Nelson, J. Service-Oriented Software Reengineering: Bertie3 as Web Services. Proceedings of the IEEE International Conference on Web Services (ICWS), Orlando, FL, USA, 11-15 July, 2005, 837–838. https://doi.org/10.1109/ICWS.2005.109
11. Cuadrado, F., García, B., Due-as, J., Parada, H. A Case Study on Software Evolution Towards Service-Oriented Architecture. Proceedings of the 22nd IEEE International Conference on Advanced Information Networking and Applications – Workshops (WAINA), Okinawa, Japan, 25-28 March, 2008, 1399–1404. https://doi.org/10.1109/WAINA.2008.296
12. Distante, D., Tilley, S. Conceptual Modeling of Web Application Transactions: Towards a Revised and Extended version of the UWA Transaction Design Model. Proceedings of the 11th International Multimedia Modelling Conference, Melbourne, Australia, 12-14 January, 2005, 439–445. https://doi.org/10.1109/MMMC.2005.28
13. Distante, D., Tilley, S., Canfora, G. Towards a Holistic Approach to Redesigning Legacy Applications for the Web with UWAT+. Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR), Bari, Italy, 22-24 March, 2006, 295–299. https://doi.org/10.1109/CSMR.2006.55
14. Erl, T. SOA Principles of Service Design. Prentice Hall, 2007.
15. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
16. Galinium, M., Shahbaz, N. Case Studies: Business and Technical Perspectives in Migration of Legacy Systems to Service Oriented Architecture. ECTI Transactions on Computer and Information Technology, 2013, 7(2), 135-145. https://arxiv.org/abs/1412.7959
17. Garriga, M., Flores, A., Cechich, A., Zunino, A. Web Services Composition Mechanisms: A Review. IETE Technical Review, 2015, 32(5), 376-383. https://doi.org/10.1080/02564602.2015.1019942
18. Garzotto, F. Ubiquitous Web Applications. Proceedings of the 5th East European Conference Advances in Databases and Information Systems (ABDIS), Vilnius, Lithuania, Lecture Notes on Computer Science, 2151, Springer Berlin Heidelberg, 25-28 September, 2001, 1-1. https://doi.org/10.1007/3-540-44803-9_1
19. Kruchten, P. The 4+1 View Model of Architecture. IEEE Software, 1995, 12(6), 42-50. https://doi.org/10.1109/52.469759
20. Lewis, G., Morris, E., Smith, D. Analyzing the Reuse Potential of Migrating Legacy Components to a Service-Oriented Architecture. Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CMSR), Bari, Italy, 22-24 March, 2006, 15-23. https://doi.org/10.1109/CSMR.2006.9
21. Lewis, G., Morris, E., Smith, D., O'Brien, L. Service-Oriented Migration and Reuse Technique (SMART). Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP), Budapest, Hungary, 24-25 September, 2005, 222–229. https://doi.org/10.1109/STEP.2005.24
22. Mateos, C., Zunino, A., Misra, S., Anabalon, D., Flores, A. Migration from COBOL to SOA: Measuring the Impact on Web Services Interfaces Complexity. Proceedings of the 23rd International Conference on Information and Software Technologies (ICIST), Druskininkai, Lithuania, October 12-14, 2017, 266-279. Communications in Computer and Information Science 756, Springer. https://doi.org/10.1007/978-3-319-67642-5
23. Mateos, C., Zunino, A., Misra, S., Anabalon, D., Flores, A. Managing Web Service Interface Complexity via an OO Metric-based Early Approach. CLEI Electronic Journal, 2017, 20(3), paper #2. https://doi.org/10.19153/cleiej.20.3.2

24. Millard, D., Howard, Y., Chennupati, S., Davis, H., Jam, E., Gilbert, L., Wills, G. Design Patterns for Wrapping Similar Legacy Systems with Common Service Interfaces. Proceedings of the European Conference on Web Services (ECOWS), Zurich, Switzerland, 4-6 December, 2006, 191-200. https://doi.org/10.1109/ECOWS.2006.14
25. O'Brien, L., Smith, D., Lewis, G. Supporting Migration to Services Using Software Architecture Reconstruction. Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP), Budapest, Hungary, 24-25 September, 2005, 81-91. https://doi.org/10.1109/STEP.2005.29
26. Ordiales, J., Mateos, C., Crasso, M., Zunino, A. Refactoring Code-First Web Services for Early Avoiding WSDL Anti-Patterns: Approach and Comprehensive Assessment. Science of Computer Programming, 2014, 89, Part C, 374-407. https://doi.org/10.1016/j.scico.2014.03.015
27. Ouni, A., Kessentini, M., Inoue, K., Ó Cinnéide, M. Search-Based Web Service Antipatterns Detection. IEEE Transactions on Services Computing, 2017, 10(4), 603-617. https://doi.org/10.1109/TSC.2015.2502595
28. Paradauskas, B., Laurikaitis, A. Business Knowledge Extraction from Legacy Information Systems. Information Technology and Control, 2006, 35(3), 214-221. http://itc.ktu.lt/index.php/ITC/article/view/11772
29. Razavian, M., Lago, P. A Systematic Literature Review on SOA Migration. Journal of Software: Evolution and Process, 2015, 27(5), 337-372. https://doi.org/10.1002/smr.1712
30. Rodriguez, J., Crasso, M., Mateos, C., Zunino, A., Campo, M. Bottom-up and Top-down COBOL System Migration to Web Services. IEEE Internet Computing, 2013, 17(2), 44-51. https://doi.org/10.1109/MIC.2011.162
31. Rodriguez, J., Crasso, M., Mateos, C., Zunino, A., Campo, M., Salvatierra, G. The SOA Frontier: Experiences with 3 Migration Approaches. In Ionita, A., Litoiu, M., Lewis, G. (Eds.), Migrating Legacy Applications: Challenges in Service-Oriented Architecture and Cloud Computing Environments, 2013, Chapter 6, 126-152. IGI Global. https://doi.org/10.4018/978-1-4666-2488-7.ch006
32. Rodriguez, J., Crasso, M., Zunino, A., Campo, M. Improving Web Service Descriptions for Effective Service Discovery. Science of Computer Programming, 2010, 75(11), 1001-1021. https://doi.org/10.1016/j.scico.2010.01.002
33. Rodriguez, J., Mateos, C., Zunino, A. Assisting Developers to Build High-Quality Code-First Web Service APIs. Journal of Web Engineering, 2015, 14(3-4), 251-285, Rinton Press Inc. http://www.rintonpress.com/journals/jwe/abstractsJWE14-34.html
34. Salvatierra, G., Mateos, C., Crasso, M., Zunino, A., Campo, M. Legacy System Migration Approaches. IEEE Latin America Transactions, 2013, 11(2), 840-851. https://doi.org/10.1109/TLA.2013.6533975
35. Smith, D. Migration of Legacy Assets to Service-Oriented Architecture Environments. Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE), Minneapolis, MN, USA, 20-26 May, 2007, 174-175. https://doi.org/10.1109/ICSECOMPANION.2007.48
36. Sneed, H. Integrating Legacy Software into a Service Oriented Architecture. Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR), Bari, Italy, 22-24 March, 2006, 3-14. https://doi.org/10.1109/CSMR.2006.28
37. Sneed, H. Wrapping Legacy Software for Reuse in a SOA. Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI), Passau, Germany, 20-22 February, 2006, 2, 345-360. GITO-Verlag Berlin.
38. Sneed, H., Sneed, S. Creating Web Services from Legacy Host Programs. Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE), Amsterdam, Netherlands, 22-22 September, 2003, 59-65. https://doi.org/10.1109/WSE.2003.1234009
39. The OSGi™ Alliance. About the OSGI Service Platform. Technical Whitepaper, Revision 4.1, 7 June 2007, 1-20. https://www.osgi.org/wp-content/uploads/OSGiTechnicalWhitePaper1.pdf
40. Yourdon, E., Constantine, L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Inc., 1st edition, 1979.
41. Zhang, B., Bao, L., Zhou, R., Hu, S., Chen, P. A Black-Box Strategy to Migrate GUI-Based Legacy Systems to Web Services. Proceedings of the IEEE International Symposium on Service-Oriented System Engineering (SOSE), Jhongli, Taiwan, 18-19 December, 2008, 25-31. https://doi.org/10.1109/SOSE.2008.8
42. Zhang, Z., Liu, R., Yang, H. Service Identification and Packaging in Service Oriented Reengineering. Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE), Taipei, Taiwan, July 14-16, 2005, 620-625.
43. Zillmann, C., Winter, A., Herget, A., Teppe, W., Theurer, M., Fuhr, A., Horn, T., Riediger, V., Erdmenger, U., Kaiser, U., Uhlig, D., Zimmermann, Y. The SOAMIG Process Model in Industrial Applications. Proceedings of the 15th IEEE European Conference on Software Maintenance and Reengineering (CSMR), Oldenburg, Germany, 1-4 March, 2011, 339-342. https://doi.org/10.1109/CSMR.2011.4