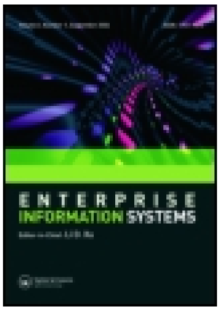

Enterprise Information Systems
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/teis20

Measuring the impact of the approach to migration in the quality of web service interfaces

Cristian Mateos [a b], Marco Crasso [a b], Juan M. Rodriguez [a b], Alejandro Zunino [a b] & Marcelo Campo [a b]

[a] ISISTAN Research Institute - UNICEN, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina

[b] CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas), Av, Rivadavia 1917, C1033AAJ, Buenos Aires, República Argentina

Version of record first published: 05 Sep 2012

Measuring the impact of the approach to migration in the quality of web service interfaces

Cristian Mateos[a,b]*, Marco Crasso[a,b], Juan M. Rodriguez[a,b], Alejandro Zunino[a,b] and Marcelo Campo[a,b]

[a]*ISISTAN Research Institute – UNICEN, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina;* [b]*CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas), Av. Rivadavia 1917, C1033AAJ, Buenos Aires, República Argentina*

(*Received 15 February 2011; final version received 28 July 2012*)

There is a good consensus on the strategic value of service-oriented architecture (SOA) as a way of structuring systems, and a common trend is to migrate legacy applications that use outdated technologies and architectures to SOA. We study the effects in the resulting Web Service interfaces of applying two traditional migration approaches combined with common ways of building services, namely, direct migration with code-first and indirect migration with contract-first. The migrated system was a 35-year-old COBOL system of a government agency that serves several millions of users. In addition, we provide a deep explanation of the trade-offs involved in following either combinations. Results confirm that the 'fast and cheap' approach to move into SOA, which is commonplace in the industry, may deliver poor service interfaces, and interface quality is also subject to the tools supporting the migration process.

Keywords: services-oriented architectures; web services; legacy system migration; direct migration; indirect migration; service modelling; service interface design; code-first; contract-first

1. Introduction

Undoubtedly, migrating legacy systems to service-oriented architecture (SOA) is an important problem of interest to the IT community. In the following subsections, we explore the complexities inherent to this problem by discussing why legacy migration is necessary, what approaches to modernise legacy systems when targeting SOA have been proposed in the literature so far and what are the implications on the quality of service interfaces. Lastly, we describe the contributions of this paper regarding this problem.

1.1. The necessity of legacy systems migration

Legacy systems are most of the time an undesired yet unavoidable reality for many enterprises. By definition, a legacy system is functioning software still actively being used, but implemented with outdated design criteria and technologies. The most representative example of legacy systems are COBOL programs, which run in

*Corresponding author. Email: cmateos@conicet.gov.ar

ISSN 1751-7575 print/ISSN 1751-7583 online
© 2012 Taylor & Francis
http://dx.doi.org/10.1080/17517575.2012.717234
http://www.tandfonline.com

mainframes and commonly implement business logic belonging to financial domains, such as bank or insurance. According to Gartner consulting,[1] over 200 billion lines of operative COBOL code are still running worldwide. Unfortunately, COBOL systems usually force organisations to incur high costs including payments for processing power, difficulty of hiring not-so-expensive programmers that master the involved technologies and lack of productive development tools. In this sense, modernisation or migration of such systems becomes a necessity. However, this has also inherent costs since, apart from the required budget, several teams with different technological skills must interact and collaborate to achieve the shared goal. This is not an easy task because technology places a language – and sometimes conceptual – barrier between the members of these teams. Thus, the decision of migrating a legacy system depends on whether the modernisation cost justifies the revenue expected from the new system.

1.2. Targeting SOA and web services for migrating legacy systems

Another important decision concerning legacy system modernisation is the choice of the target programming paradigm, and hence the technologies to be used for designing and implementing the new system. Nowadays, for achieving this, SOA has been adopted in many cases (Bichler and Lin 2006, Erickson and Siau 2008, Mietzner *et al*. 2011). Service-oriented architecture (SOA) has mainly evolved from component-based software engineering by introducing a new kind of building block called *service*, which represents a capability offered by an organisation to one or more service consumers. In SOA, services can belong to two complementary classes: *business services* and *software services* (Kohlborn *et al*. 2009). A *business service* is a set of actions characterising an organisation (core business) that are exposed via well-defined operations. A *software service* is the specific part of an application or system that actually allows other applications to invoke a business service. Service-oriented architecture (SOA) is in general the right target for migration when loose coupling among applications and software services, and agility to respond to changes in requirements are needed.

The commonest technological choice for materialising SOA designs is Web Services. The term Web Services refers to a technology stack for enabling programmes with well-defined interfaces to be described and consumed by means of ubiquitous protocols (Erickson and Siau 2008, D'Mello and Ananthanarayana 2010, Gong *et al*. 2010, Wang *et al*. 2010), usually Web Service Description Language (WSDL) plus XML Schema Definition (XSD) and Simple Object Access Protocol (SOAP) (Gudgin *et al*. 2007), respectively. A service provider, such as a business or a governmental organisation, provides meta-data for a service, including a specification of its functionality in WSDL (Erl 2007). Web Service Description Language (WSDL) allows providers to specify a service interface as a set of abstract operations with inputs and outputs, and to specify the associated binding information so that consumers can invoke the offered operations. Inputs and outputs have associated data types specified in XSD (Gao *et al*. 2009). Then, service consumers (e.g. third-party applications) can invoke Web Service operations by interpreting the contents of the WSDL document associated to the corresponding service, and exchanging structured information through SOAP.

Basically, a legacy system can be migrated to exploit the SOA paradigm using two approaches (Li *et al*. 2007), namely, wrapping the legacy system or reimplementing it.

The first method, called direct migration, consists in writing a new software component that exposes selected existing software components as services. In contrast, the other method, known as indirect migration, requires re-implementing all the functionality that will be exposed as individual services within the new SOA platform. Clearly, these migration approaches are radically different. In fact, other alternative combinations are possible, such as reengineering a portion of the legacy system, while keeping other portions untouched. An example of this could be re-implementing only the COBOL code, but keeping the 'old' database as is.

1.3. Implications of approaches to legacy systems migration on WSDL interface design

Having well-designed and descriptive WSDL documents is a mandatory requirement to successful SOA. But unless appropriately specified by its provider, a WSDL document can be counterproductive and obscure the purpose of a service. This makes the service difficult to be understood from a functional perspective, which in turn raises consumer application development costs. Although the literature (Crasso *et al.* 2010) has already acknowledged this problem, most organisations still follow the direct migration approach alongside a somewhat fast WSDL document generation method known as *code-first* (Mateos *et al.* 2010). Basically, this means that WSDL documents are not directly created by developers but are instead automatically derived from the source code implementing the corresponding software services. The drawback of this approach is that resulting WSDL documents are hard to deal with as they are negatively influenced by bad implementation practices in the code, and the inefficacy of WSDL generation tools. With respect to the latter, some of these tools replace argument names with non-explanatory names like 'arg0', 'arg1' during the generation process, or just do not include descriptive comments present at either methods or classes headers into the WSDL documents, or generate redundant XSD code for defining shared data types.

On the other hand, by basing on the fact that developers of client applications tend to prerer properly designed WSDL documents over those having the problems mentioned in the previous paragraph (Rodriguez *et al.* 2010c), it is reasonable to expect that when service developers have full control over their WSDL documents, they build WSDL interfaces better designed than those automatically generated. Although the SOA community has the reasonable suspicion that indirect migration results are better in terms of WSDL documents quality (Papazoglou and van den Heuvel 2006), until now, an open research question is how much better the WSDL documents resulting from indirect migration are. The lack of evidence about the implications of the approach to migration on WSDL interfaces design forces software engineers to choose between one migration approach or another by basing on other classical criteria, while disregarding the one that directly impacts on the final SOA system frontier. In this context, a SOA frontier is the set of WSDL documents that describe a service-oriented system. Figure 1 depicts how a SOA frontier allows the interaction between not only service providers and service consumers, but also service providers and service registries. The most important reason for the lack of objective information about the impact of both direct and indirect migration on WSDL and hence SOA frontier quality is that it is extremely rare to find a system on which both approaches have been applied.

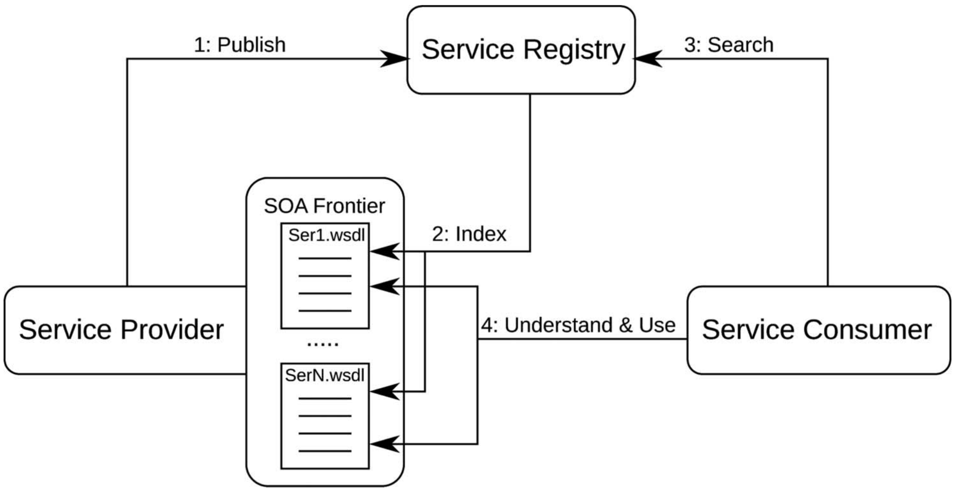

Figure 1. SOA frontier and SOA roles.

1.4. Contributions of this paper

This paper analyses the outcomes of migrating a 35-year-old legacy COBOL system using direct migration and indirect migration approaches, with two different WSDL construction methods. The system is owned by an Argentinean government agency. The goal of the migration process was to allow agency website developers to integrate some of the core functionality with the agency Web portal for improving the e-government model and, in turn, to expand the accessibility to other governmental agencies.

Due to some budget constraints at the time the project started and the urgency of improving its Web portal, the agency opted for a direct migration targeting .NET as the service platform. Although the Web Services could be used to improve the portal, the complex interfaces resulting from the migration approach rendered the consumption of these services outside of the agency rather hard. Once the most urgent necessity was fulfilled, the agency needed to make their system more reusable. Therefore, the agency started a project with our University to migrate the system in a more friendly way for third-parties developers, i.e. exposing better designed service interfaces. As a result of this project, a second attempt to migrate the system was performed. This attempt consisted in using indirect migration for re-implementing in .NET part of the same system.

As a consequence of having both migration methods using the same technology, it was possible to compare them with respect to the trade-off between resulting SOA frontier quality and migration costs. We measured the impact of both approaches in the obtained service interfaces by basing on extensive research on WSDL design best practices (Crasso *et al.* 2010, Rodriguez *et al.* 2010a, 2010c). Besides showing that there is an unquestionable trade-off between the effort one puts on (and the cost of) migration and the design quality of resulting SOA frontiers, our results provide objective means to quantitatively analyse service interfaces in terms of understandability, discoverability and clarity.

The rest of the article is structured as follows. Section 2 presents related efforts. Section 3 describes the aforementioned project and its two migration attempts.

Section 4 reports a comparison of the SOA frontiers obtained by either attempts. Finally, Section 5 presents the conclusions and future research.

2. Related work

Legacy system migration has been always recognised as a tempting but challenging endeavour. In the past few years, migration from mainframe systems to service-oriented ones has attracted the attention of both the industry and the academia. Governments increasing tendency to adhere to open-source platforms has also encouraged migration. Below, we summarise the most relevant studies about defining migration approaches, reporting migration experiences and comparing migration results.

Migrating a software system refers to the process of moving the system to a different target environment, while preserving the original system functionality and data. Specifically, there are two main approaches to migrating legacy systems to SOA, namely, *direct migration* and *indirect migration* (Li *et al*. 2007). The former can be seen as a bottom-up approach because it encourages engineers to *servify*, i.e. to expose as services, the original components of the legacy system. Accordingly, final services potentially inherit the interfaces, and so the offered functionality, from the older components. Instead, indirect migration requires to abstractly define target service interfaces based on a high-level view of the functionality of the legacy components, and then inspecting/re-writing its source code, or parts of it, for implementing the defined service interfaces. Alternatively, when the definition of service interfaces is the result of comparing desired high-level views of the functionality and the functionality that is actually offered by the legacy system, one arrives at the settlement of *meet-in-the-middle* service interfaces (Ricca and Marchetto 2009). In this sense, *meet-in-the-middle* refers to an approach to define the SOA frontier of a migrated system, which is commonly used within indirect migration attempts.

There are different strategies for implementing each approach. The most frequent strategies are *wrapping* and *re-engineering*. The wrapping strategy is based on providing an SOA-enabled interface to existing components in order to expose them as services to other software components. This strategy is commonly used when re-writing legacy source code is too expensive, and instead a fast, cost-effective solution is needed (Almonaies *et al*. 2010). On the other hand, re-engineering involves re-thinking, restructuring, redesigning and re-implementing the legacy system to transform it into a well-shaped, service-oriented system (Almonaies *et al*. 2010).

Different researches have reported migration experiences, for example Li *et al*. (2007), De Lucia *et al*. (2008) and Colosimo *et al*. (2009). The work shown in Li *et al*. (2007) not only reports an experience, but also proposes a systematic meet-in-the-middle approach to migration. This approach consists of five steps. First, high-level diagrams – data flow diagrams (DFD), entity relationship diagrams (ERD), etc. – of the original system are utilised to re-engineer its business objects. Second, system business processes are analysed from these diagrams. The resulting business objects and processes are integrated into *business services* at the third step of the proposed approach. Then, *software services* are defined to realise the business services using SOA. Finally, the fifth step is for implementing the target SOA using Web Services standards. This approach is a meet-in-the-middle one, since at the third step business objects (re-engineered at step 1) meet needed business services (derived at step 2).

In De Lucia *et al.* (2008), the authors analyse the results of performing a direct migration, through wrapping, of a legacy system back-end, and an indirect migration, via re-engineering, of its front-end. A tool to support the migration process is presented by the authors as well. Likewise, Colosimo *et al.* (2009) propose an integrated development environment (IDE) for the easy migration of legacy COBOL programs to Web systems. The authors compared the results obtained when employing their IDE vs. not using it by migrating a case study. The reported comparison is in terms of developer's productivity.

One major difference between the works mentioned in the previous two paragraphs and ours is that this paper reports two migration experiences to SOA of the same legacy system by comparing migration results in terms of service interfaces quality. Broadly, as far as we know, there is not even a consensus about how to quantitatively compare two migrated versions of a legacy system. In this paper, we focus on comparing the service interfaces of each evaluated version by means of WSDL-level metrics. This is because service interfaces play the most important role in enabling third-party developers to understand, discover and reuse services (Crasso *et al.* 2010).

When using Web Services to materialise service-oriented systems, designing service interfaces requires employing WSDL. Several important concerns, such as granularity, cohesion, discoverability and reusability, should influence design decisions to result in clear WSDL document designs (Papazoglou and van den Heuvel 2006). Therefore, two WSDL documents can be compared based on the aforementioned concerns. In Rodriguez *et al.* (2010c), the authors identified and built a catalogue of frequent practices that attempt against these concerns in publicly available WSDL documents. This catalogue presents the bad practices as anti-patterns each comprising a name, a symptom description and a sound solution. Moreover, the article offers empirical evidence of the advantages of avoiding the identified anti-patterns for improving the chance of services to be understood and reused. Then, the catalogue of anti-patterns can be used to compare two WSDL-based service interfaces. Specifically, one could account anti-pattern occurrences within a set of given service interfaces because the fewer the occurrences are, the better the resulting WSDL documents are. In this sense, as a collection of WSDL-described services is the outcome of the aforementioned approaches for moving from mainframe environments into service-oriented ones, the catalogue of WSDL anti-patterns was used in this paper to assess the quality of the resulting service interfaces in our reported experience. Moreover, this approach might allow software practitioners to compare among different migration alternatives in similar scenarios.

3. The project

This section describes a project comprising two migration attempts: one for migrating an entire legacy system to SOA using a direct migration approach by wrapping it, and another for migrating the most accessed parts of such system using an indirect migration by re-engineering it. The section is organised as follows. Section 3.1 presents an overview of architectural and technological aspects of the project. Section 3.2 describes the characteristics of the direct migration attempt, including employed method, tools and technologies. The indirect migration attempt is described in Section 3.3. Finally, Section 3.4 summarises the obtained results from both attempts.

3.1. Architectural and technological description of the legacy system under study

The project under study involved an initiative to modernise the information system of an Argentinean government agency. Roughly, the system is data centric and is composed of several subsystems that maintain data records related to individuals including complete personal information, relationships, work background, received benefits and so forth. The system is written in COBOL, runs on an IBM AS/400 mainframe and accesses a DB2 database with around 0.8 PetaBytes. On the other hand, there are some COBOL programs accessing historic data through Virtual Storage Access Method (VSAM), a storage and data access method featured by a number of IBM mainframe operating systems. Virtual Storage Access Method (VSAM) relies on a record-oriented data allocation scheme at the file-system level. Moreover, some of the COBOL programs are only accessed through an intra-net via 3 270 terminal applications, while other programs are grouped in Customer Information Control System (CICS) transactions and consumed by Web applications. Customer Information Control System (CICS) is a transaction manager designed for rapid, high-volume processing, which allows organising a set of programs as an atomic task. In this case, these programs consist of business logic and database accesses, mostly input validations and queries, respectively.

For the sake of illustration, a brief overview of only six transactions is shown in Table 1, which indicates the number of non-commented source code lines,[2] the number of SQL queries performed, and the number of lines and files associated with a transaction. When a program P_1 calls a program P_2, or imports a Communication Area (COMMAREA) definition C, or includes a SQL definition S, it is said that P_1 is associated with P_2, or C, or S, respectively. On average, each program had 18 files, comprised 1803 lines of code and performed six SQL SELECT statements.

Note that all the CICS/COBOL transactions basically receive an input, perform some queries to the database, apply some analysis to these query results and return the requested information. This means that all the interactions between the system and client applications are initiated by the latter. As a result, to expose these transactions as Web Services, it is only necessary to use request–response WSDL operations. This is important because the Web Service Interoperability (WS-I) standards[3] only allows two types of operations being one of these the request–response operations.

Table 1. Characteristics of most important project transactions according to the mainframe load in terms of executed transactions during January 2010.

Program	SQL	COMMAREA(s)		Include(s)	
Number of lines	Number of queries	Number of lines	Number of files	Number of lines	Number of files
265	2	518	7	683	6
416	2	1114	6	141	5
537	3	10	2	800	29
1088	10	820	4	580	16
543	10	820	4	411	10
705	10	956	6	411	10

3.2. First attempt: direct migration (wrapping)

The first migration attempt took place during 2009 and involved the exposure of most of the CICS transactions through Web Services. The reason behind this initiative was basically the inception of a program by the Argentinean government designed to improve the IT landscape of the government organisations as well as supplying citizens better access to information and technology. Another goal was to replace other access methods to the backend functionality (e.g. 3270 terminal application) with Web Services because giving a single technological entry point was expected to facilitate system administration as well as client application building. Due to budget constraints and some strict deadlines partially imposed by this program, a direct migration to .NET Web Services was carried out. To some extent, this allowed the IT department of the agency to gain understanding of the capabilities of some portions of the system that had been developed many years ago by external software professionals.

3.2.1. Methods and tools employed

Methodologically, the IT department members followed for each migrated transaction a wrapping strategy (Almonaies *et al.* 2010) that comprised four steps:

(1) Automatically creating a COM+ object including a method with the inputs/outputs defined in the associated COMMAREA, which forwards invocations to the transaction. This was done by using a tool called COMTI Builder (Leinecker 2000).
(2) Automatically wrapping the COM+ object with a C# class having only one method that invokes this object by using Visual Studio.
(3) Manually including specific annotations in the C# code to deploy it and use the framework-level services of the .NET platform for generating the WSDL document and handling SOAP requests.
(4) Testing the communication between the final Web Service and its associated transaction. This was performed by means of a free tool called soapUI (http://www.soapui.org).

To clarify these four steps, a word about the employed technologies and tools is needed. A COMMAREA is a fixed region in the RAM of the mainframe that is used to pass data from an application to a transaction. Conceptually, a COMMAREA is a C++ struct with (nested) fields specified by using native COBOL data types. Component Object Model Transaction Integrator (COMTI) is a technology that allows a transaction to be wrapped with a Component Object Model Plus (COM+) object. The tool named COMTI Builder receives a COMMAREA as input to automatically derive a type library (TLB), which is accessible from any component of the .NET framework as a COM+ object afterwards. Component Object Model Plus (COM+) is an extension to COM that adds a new set of functions to introspect components at run-time. Finally, the soapUI testing tool receives one or more WSDL documents and automatically generates a client for the associated service(s), which allows the generation of test suites.

Basically, each step, but step 4, adds an onion layer to the original transactions. In this context, the Wrapper Design Pattern is central to the employed steps, since wrapping consists in implementing a software component interface by reusing existing components, which can be any of a batch program, an on-line transaction, a program, a module, or even just a simple block of code. Wrappers not only implement the interface that newly developed objects use to access the wrapped systems, but also are responsible for passing input/output parameters to the encapsulated components. Then, from the inner to the outer part of a final service, by following the described steps the associated transaction was first wrapped with a COMTI object, which in turn was wrapped by a COM+ object, which finally was wrapped by a C# class that in the end was offered as a Web Service. To do this, implementation classes were deployed as .NET ASP 2.0 Web Service Applications, which used the framework-level services provided by the .NET platform for generating the corresponding WSDL document and handling SOAP requests. As the reader can see, the SOA frontier was automatically derived from the C# code, which means that WSDL documents were not made by human developers but they were automatically generated by the .NET platform. In other words, the employed WSDL document construction method was *code-first*.

3.2.2. Overview of costs and results

As a result of this migration attempt, the IT department obtained an individual Web Service for each migrated transaction. The WSDL documents of these services presented some threats to reusability. Almost all of the services comprised a lot of business logic and ca. 100 output parameters; there were coarse-grained services in the sense they offered a large view of the back-end data; there were services offering almost exactly the same functionality; their associated descriptions had no documentation at all.

Figure 2 summarises the anatomy of the resulting Web Services. Each Web Service consisted of two main parts: (1) a thin C# tier running on the .NET platform and (2) the original CICS/COBOL transactions and programs running on the IBM mainframe.

As reported by the organisation's IT department, it took one day to train a developer on these four steps and the three tools employed, namely, COMTI Builder, Visual Studio and soapUI.[4] Then, trained developers migrated one transaction per hour, mostly because all the steps but one (step 3) were tool-supported and automatic. Since the agency had the respective software licenses for COMTI Builder and Visual Studio tools beforehand, choosing them was a harmless decision from an economical viewpoint. Appendix 1 presents an example useful for highlighting the fact that the combination of the direct migration approach and the wrapping strategy along with the tool-set and method (i.e. code-first) employed for implementing them, allowed the IT department members to migrate the target transactions with few resources. Indeed, the reader should recall that this migration attempt was characterised by tight budget constraints and strict deadlines.

3.3. Second attempt: indirect migration (re-engineering)

On February, 2010, the IT department outsourced the migration of 32 top priority transactions ranked by usage history using an indirect migration approach, which involved re-thinking, re-designing and re-implementing the transactions to transform them into a well-shaped, service-oriented system. In other words, this means that the attempt consisted in migrating the system using a conventional indirect migration

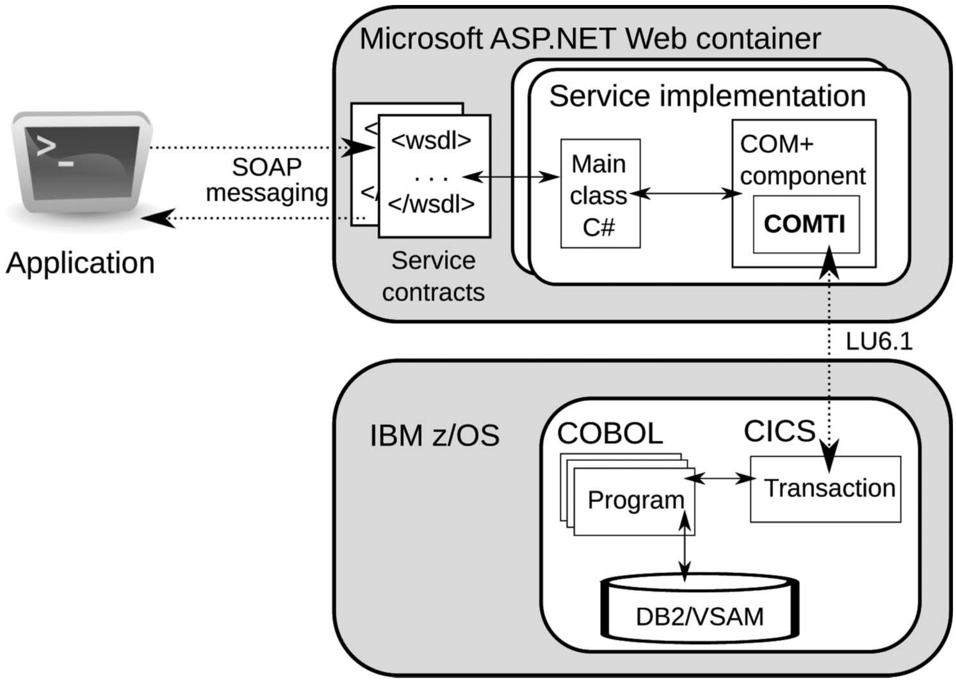

Figure 2. Anatomy of the Web Services that resulted from the first migration attempt.

approach, which is methodologically described in Li *et al*. (2007). In part, the goal was to remove these prioritised transactions from the mainframe so as to alleviate CPU load. The starting point of this attempt was the business code of the prioritised transactions, i.e. the starting point was the same for both migration attempts.

3.3.1. Methods and tools employed

Methodologically, this attempt was performed following the indirect migration approach (Li *et al*. 2007). This approach basically implied following five steps:

(1) Manually defining potential WSDL documents based on the knowledge, the agency had on the interface and functionality of the original transactions. For each service operation, a brief explanation using WSDL documentation elements was included.
(2) Exhaustively revising the legacy source code.
(3) Manually refining the WSDL documents defined during (1) by basing on opportunities to abstract and reuse parameter data type definitions, group functionally related transactions into one cohesive service, improve textual comments and remove duplicated transactions, which were detected at step 2. For data type definitions, we followed best practices for naming type elements and constraining their ranges.
(4) Supplying the WSDL documents defined at (3) with implementations using .NET.
(5) Testing the migrated services with the help of the agency IT department.

During the first step, three specialists on Web Services technologies designed preliminary WSDL documents, based on the knowledge the agency had on the functionality of the transactions, to sketch the desired system's frontier together. This step comprised daily meetings not only between the specialists and the project managers in charge of the original COBOL programs, but also between the specialists and the project managers responsible for developing client applications that would consume the migrated Web Services. Unlike the code-first approach, in which service interfaces are derived from their implementations, the three specialists used *contract-first*, which encourages designers to first derive the technical contract of a service using WSDL, and then supply an implementation for it. Usually, this approach leads to WSDL documents that better reflect the business services of an organisation, but it is not commonly used in the industry since it requires WSDL specialists. This step might be carried out by defining service interfaces in C# and then using code-first for generating WSDL documents, specially when analysts with little WSDL skills are available.

The second step involved revising the transactions code with the help of documents specifying functionality and diagrams illustrating the dependencies between the various transactions to obtain a bigger picture. This was done to output a high-level analysis of the involved business logic, since the existing COBOL to some extent conditioned the functionality that could be offered by the resulting services. The selected transactions comprised 261,688 lines of CICS/COBOL code (600 files). Six software analysts exhaustively revised each transaction and its associated files under the supervision of the specialists during three months. Once the complete big picture of the transactions was obtained, it was used for refining the previously obtained WSDL documents.

The third step consisted in refining the WSDL documents obtained in the first step by basing on the output of the second step. Broadly, we abstracted and reused parameter data type definitions, grouped functionally related transactions into one cohesive service, improved textual comments and names and removed duplicated transactions. For data type definitions, we followed best practices for naming data type elements and constraining their ranges. From this thorough analysis, we derived potential interfaces for the target services and a preliminary XSD schema document subsuming the entities implicitly conveyed in the original COMMAREA definitions. In this sense, we iteratively built the final service interfaces based on the desired business services, which impact on the implementation of services, as well as the interfaces derived from the existing CICS/COBOL code, which to some extent condition the functionality that can be exposed by the resulting software services. Note that to some extent, this migration attempt might be seen as an application of the *meet-in-the-middle* approach (Ricca and Marchetto 2009). However, since both migration attempts are analysed from the CICS/COBOL point of view only, this particular attempt used indirect migration because all the code was replaced by new C# code (Li *et al.* 2007).

The fourth step regarded re-implementing the services and began once the WSDL documents were defined. Two more people were incorporated in the project for implementing the services using the .NET ASP 2.0 Web Service Application template as required by the agency. Hence, the three specialists trained eight software developers in Visual Studio 2008, C# and a data mapper, called MyBatis.[5] This library frees developers from coding typical conversions between database-specific data types and programming language-specific ones. MyBatis connects to DB2 mainframe databases using IBM's DB2Connect,[6] an infrastructure for connecting Web, Windows, UNIX, Linux and mobile applications to z/OS and AS/400 back-end data. It is worth noting that to bind the defined WSDL documents with their .NET implementations, we had to extend the ASP 2.0 'httpModules' support. Concretely, the three specialists developed a custom module that returns a manually specified WSDL document to applications, instead of generating it from source code, which is the default behaviour of .NET.

The fifth step was to test the resulting services with the help of the agency IT department. Basically, each new Web Service was compared to its CICS/COBOL counterpart(s) to verify that with the same input the same output was obtained. If some inconsistency between the new services and the old CICS/COBOL system was detected, the services were revised and re-tested. This step was repeated until the agency IT department had the confidence that the new SOA-based system was as good as the old system from a functional point of view.

3.3.2. Overview of costs and results

In the end, the indirect migration attempt ended up with seven Web Services having 45 operations in total. The WSDL documents of these services were designed by taking into consideration service interface design best practices. For instance, services had comments for describing offered operations, but not business object definitions, which were placed in a separate XSD file so they can be reused from different service descriptions.

Figure 3 depicts an overview of the anatomy of a migrated service. Each service implementation consisted of a WSDL document and a handful of C# classes. A service main class validates inputs and calls data mapper classes to gather the information requested by client applications.

Regarding the costs of this migration attempt, monetarily, it cost the agency 312,000 US dollars. Table 2 details the human resources involved in the second attempt of the project. All in all, it took one year plus one month for six software analysts, two more developers incorporated at step 4 and three specialists to migrate 32 priority transactions. It is worth noting that no commercial tools were needed, apart from the IDE for which the agency already had licenses. Indeed, the cost of indirectly migrating the entire legacy system is much higher than the cost of applying a direct migration.

3.4. Summary

The entire project involved an initiative to SOA-enable a real software system comprising a set of data centric CICS/COBOL transactions and programs that run on a mainframe. The project comprised two attempts. In the first attempt, the IT department members of the agency employed a *direct migration by wrapping* that ended with all the original transactions being called from Web Services, whose WSDL documents were constructed using the code-first method. The next year, the second migration attempt of the project started. In this attempt, we were hired to migrate a subset of the transactions comprising 261,688 lines of source code by following the *indirect migration* with a re-engineering strategy, plus the contract-first methodology to build the WSDL documents of the resulting services.

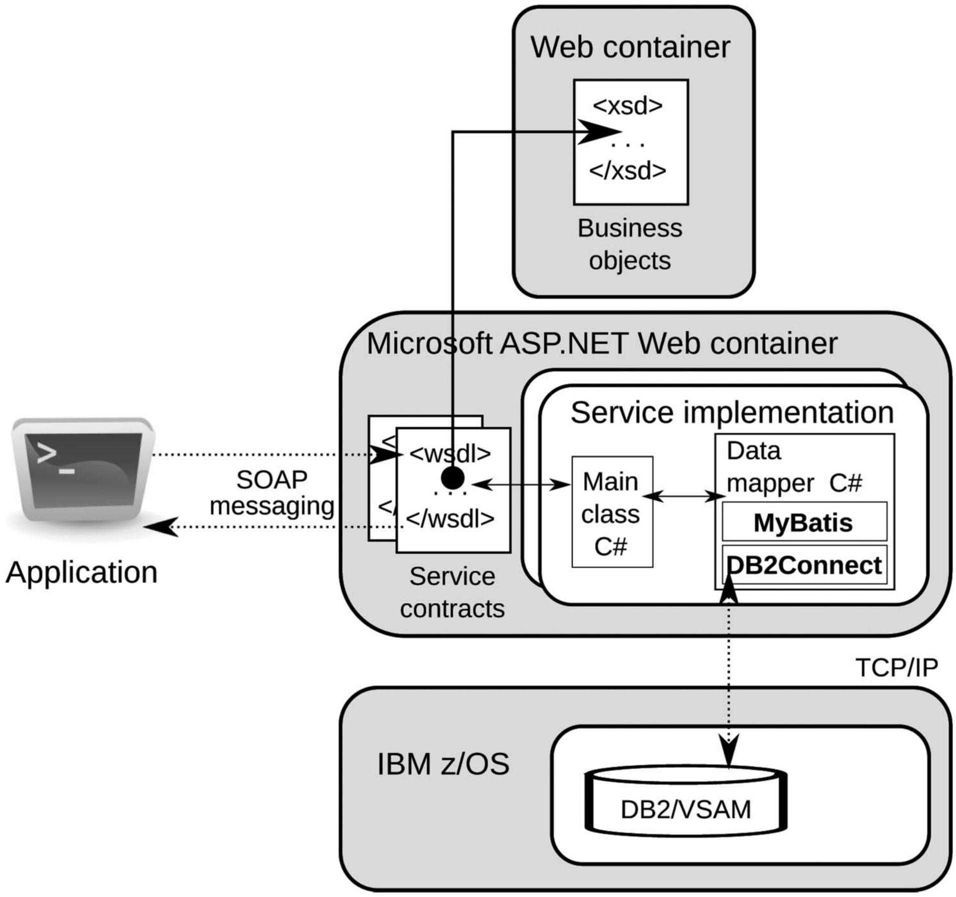

Figure 3. Anatomy of the Web Services that resulted from the second migration attempt.

Table 2. Required manpower over months.

Step	People	Role	Time (in months)
1	3	WSDL specialists	1
2	6	Software analysts	3
3	3	WSDL specialists	1
4	8	Software developers	6
5	8	Software developers	2

There were remarkable differences between the results of both migration attempts when we talk about costs and the level of mainframe independence achieved. For example, taking into account that it took one day to train a developer and in turn it took one hour/developer on average to migrate a transaction with the first attempt methodology and tools, one developer then migrates the 32 prioritised transactions in five standard eight-hour working days, instead of the 13 months the second migration attempt required. However, these months were not in vain but interestingly represented a big step toward unloading the mainframe and therefore becoming less dependent from it, and providing a single technological entry point to the system functionality.

Despite the fact that first attempt Web Services were hard to use and this motivated the second attempt, the benefits of this latter with respect to SOA system frontier quality may be not intuitive. Therefore, we analysed the Web Service interfaces that resulted from both migration attempts. A detailed comparison of the obtained results is presented in the next section.

4. Comparison of the resulting WSDL contracts

We set forth the hypothesis that when migrating a legacy system to SOA, most of service interfaces design issues may arise as a consequence of two crucial decisions, the first one regarding the methodological approach to and strategy of migration, and an operational one, i.e. concerning the WSDL document generation method and tool-set employed.

To evaluate the impact of the approach to system migration (i.e. direct or indirect) in conjunction with the WSDL construction methodology (i.e. code-first or contract-first) on the quality of Web Service interfaces, we analysed the WSDL documents that resulted from both migration attempts. To do this, we used not only classical metrics, such as total lines of code and number of resulting files, but also a well-established set of metrics for WSDL-based interfaces. These metrics are based on a synthesised catalogue of common WSDL bad practices – i.e. anti-patterns – that jeopardise WSDL understandability and legibility concerns (Rodriguez *et al.* 2010c). In this sense, we focused on anti-pattern occurrences within each collection because the fewer the occurrences are, the better the collection is. In addition, we analysed business object definitions reuse. It is worth noting that other non-functional requirements, such as performance, reliability or scalability, have been intentionally left out of this paper since we were interested in WSDL quality after migration. For confidentiality reasons, we will use general examples instead of showing real WSDL documents, though the measured results were obtained from the deployed services.

To perform the comparisons, we used as input three different data-sets of WSDL documents:

- *Direct migration*: The WSDL documents that resulted from the first attempt of the project. As such, the associated services were obtained by using direct migration and implementing wrappers to the transactions, and by using the default tool-set provided by Visual Studio 2008 that supports *code-first* generation of service interfaces from C# code.
- *Indirect migration*: The WSDL documents obtained via the indirect migration approach followed during the second attempt of the project, i.e. indirect migration and at the same time the contract-first WSDL generation method.
- *Code-first indirect migration* (or meet-in-the-middle as explained in 2): This data-set was generated from the indirect migration code, but disregarding the manually generated (i.e. contract-first) WSDL documents. In other words, these WSDL documents were generated from the re-engineered code, but by following the code-first method instead of relying on the manually created (contract-first) WSDLs. As a result, we had a similar service interface

compared to the interfaces of the services from the previous data-set, but with some potential noise introduced by the WSDL document generation tool. Therefore, we can assess the impact or effect of the employed code-first tool in the WSDL document quality. This impact is also present in the direct migration because the WSDL document was also generated automatically.

The missing combination (i.e. *Contract-First Direct Migration*) given by writing the WSDL documents and then adapting them to the transactions using .NET adapter code has not been considered because it would had produced very similar WSDL documents to that of the *Indirect Migration* data-set. In this sense, as mentioned earlier in this section, the focus of this comparison is put on WSDL document quality, whereas implementation-related metrics are not reported here. The only difference between the combination not taken into consideration and the method behind the *Indirect Migration* data-set is that the implementation of services is different. This means that instead of connecting .NET code to the back-end database by re-implementing the existing logic, .NET would be connected to the original transactions via regular C# adapters. On the other hand, this method fails at removing legacy code, which means that if there is another migration attempt, there would be two legacy and non-refactored system implementations to be considered, i.e. one mostly written in CICS/COBOL and another one partially written in .NET. Therefore, we present the results of comparing the three data-sets of WSDL documents listed above.

4.1. Classical metrics analysis

The first evident difference in the results is the number of WSDL documents generated by the different methods. The number of WSDL documents in each data-set is 32 (*Direct Migration*), 7 – plus 1 separated XSD file – (*Indirect Migration*) and 7 (*Code-First Indirect Migration*). Despite exposing the same functionality as Web Services, the *Direct Migration* data-set had more than four times WSDL documents than the *Indirect Migration* data-set in its two variants. This stems from the fact that the WSDL documents of the *Direct Migration* data-set were blindly generated for each transaction, whereas transactions were deliberately grouped by their semantic 'similarity' in the same Web Service for the WSDL documents of the two *Indirect Migration* data-sets. This sort of *functionality aggregation*, which is a well-known design principle (Yourdon and Constantine 1979), is highly desirable since it allows Web Service consumers to look for semantically-related operations within a reduced set of WSDL documents.

Another difference is the number of lines of WSDL and XSD code per document, which on average is 222, 512 and 1099, respectively. Figure 4 shows a detailed view of the resulting code lines per service interface description. Indeed, it has been shown that developers, when faced with two or more WSDL documents that are similar from a functional perspective, they tend to choose the most concisely described (Crasso *et al.* 2010). A corollary of this is that users will prioritise smaller WSDL documents over larger ones. As shown in Figure 4, the *Direct Migration* data-set contained smaller WSDL documents but at the cost of scattering the system functionality across several Web Services. Thus, finding a needed operation potentially requires to inspect more WSDL documents. On the other hand, both

Indirect Migration and *Code-First Indirect Migration* data-sets include less and but larger WSDLs, therefore arguably offering a better alternative to the contract length/functionality scattering trade-off. However, the *Indirect Migration* data-set has the same number of WSDL documents (plus one artefact including the refactored XSD definitions), while requiring around 46% less code lines.

Another important difference is the amount of documentation present in resulting WSDL documents. The total number of documentation lines is 0, 242 and 90 for the WSDL documents belonging to *Direct Migration, Indirect Migration* and *Code-First Indirect Migration* data-sets, respectively. This is all the documentation for the 32 migrated transactions, which means that there are, on average, 0, 7.5 and 2.8 documentation lines per migrated transaction.

4.2. Anti-patterns occurrences analysis

As suggested earlier, anti-patterns can be used as objective quality metrics for WSDL documents. Eight anti-patterns have been identified in Rodriguez *et al*. (2010a), six of which are simultaneously present in at least one WSDL document of the three data-sets under study. The anti-patterns that have been found are:

- Inappropriate or lacking comments (Fan and Kambhampati 2005): Some operations within a WSDL have no comments or the comments do not effectively describe their associated elements (messages, operations).
- Ambiguous names (Blake and Nowlan 2008): Some WSDL operation or message names do not accurately represent their intended semantics.

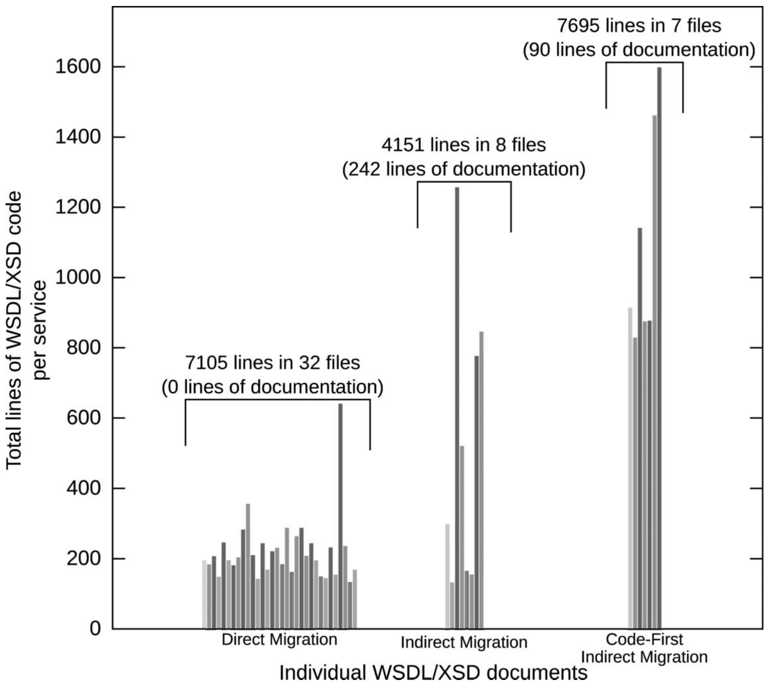

Figure 4. WSDL and XSD code lines of the different service contracts.

- Redundant port-types: A port-type is repeated within the WSDL document, usually in the form of one port-type instance per binding type (e.g. HTTP, HTTPS or SOAP).
- Enclosed data model: The data model in XSD describing input and output data types is defined within the WSDL document instead of being defined in a separate file, which makes data type reuse across several Web Services very difficult or impossible. The exception to this rule occurs when it is known before-hand that data types are not going to be reused. In this case, including data type definitions within WSDL documents allows constructing self-contained contracts, so it is said that the contract does not suffer from the anti-pattern.
- Redundant data models: A data type is defined more than once in the same WSDL document.
- Undercover fault information within standard messages (Beaton *et al.* 2008): Error information is returned using output messages rather than native SOAP Fault messages.

Table 3 summarises the results of the anti-patterns analysis. The rows represent under which circumstances the WSDL documents are affected by a particular anti-pattern. When an anti-pattern affects a portion of the WSDL documents in a data-set, we have analysed which is the difference between these WSDL documents and the rest of the WSDL documents in the same data-set. Since there are anti-patterns whose detection is inherently more subjective (e.g. *Inappropriate or lacking comments* and *Ambiguous names*) (Rodriguez *et al.* 2010b), we performed a peer-review methodology after finishing their individual measurements to prevent biases.

Achieved results show that the WSDL documents of the *Direct Migration* data-set are affected by more anti-patterns than the ones belonging to the *Indirect Migration* data-sets. The first two rows describe anti-patterns that impact on services discoverability and reusability (Crasso *et al.* 2010), and relate to documentation and naming issues. It is reasonable to expect these anti-patterns to affect the WSDL documents of the *Direct Migration* data-set, since all information included in them is derived from CICS/COBOL code, which does not offer a standard way to indicate

Table 3. Anti-patterns in the three WSDL data-sets.

Anti-pattern/ data-set	Direct migration	Indirect migration	Code-first indirect migration
Inappropriate or lacking comments	Always	Never	Never
Ambiguous names	Always	Never	Never
Redundant port-types	When supporting several protocols	Never	Never
Enclosed data model	Always	Never	Always
Undercover fault information within standard messages	Always	Never	Never, but present other issues
Redundant data models	When two operations use the same data type	Never	When two operations use the same data type

from which portions and scope of a COBOL code existing comments can be extracted and reused. At the same time, because names in CICS/COBOL have associated length restrictions (e.g. up to four characters in some CICS and/or COBOL flavours), names in the resulting WSDL documents are too short and difficult to be read.

The third row describes an anti-pattern that ties abstract service interfaces to concrete implementations, hindering black-box reuse (Crasso *et al.* 2010). We have checked that this anti-pattern was caused by the tools employed for generating WSDL documents during the first migration attempt. By default, the employed tool produces the anti-pattern. To avoid the anti-pattern, developers should provide not so well-known annotations within the C# source code implementing the service. Likewise, the fourth row describes an anti-pattern that is generated by many code-first tools, which force data models to be included within the generated WSDLs, and cannot be avoided within the WSDLs of neither the *Direct Migration* nor the *Code-First Indirect Migration* data-set.

The anti-pattern described in the fifth row of the table deals with errors being transferred as part of output messages, which for the *Direct Migration* data-set resulted from the original transactions that used the same COMMAREA for returning both output and error information. This means that when transactions are exposed as Web Services, the output message of these services convey both output and error information. In contrast, the WSDL documents of the *Indirect Migration* data-set had a proper designed error handling mechanism based on standard SOAP Fault messages. Finally, Code-First Indirect Migration WSDL documents had no error handling because the tool employed for generating them does not support Fault messages generation. As a result, the output messages in the generated WSDL documents do not convey error information. In other words, although the .NET platform allows firing this special kind of SOAP messages at the service code level, associated Fault messages are not present in generated WSDL documents. This is a problem because a service consumer might not be expecting this kind of error notifications at runtime because they were not present in the corresponding WSDL document when the service was first discovered.

The last anti-pattern relates to bad data model designs. Redundant data models usually arise from limitations or bad use of the tools employed to generate WSDL documents. Basically, tools based on code-first WSDL construction commonly force data models to be included within the generated WSDL document, which explains why the enclosed data model anti-pattern is present in both *Direct Migration* and *Code-First Indirect Migration* data-sets. Although some XSD definitions are in the WSDL documents resulting from *Indirect Migration*, the anti-pattern is not considered as present because the data types that are shared by the services are imported from a separate file. As discussed in Rodriguez *et al.* (2010c), this situation does not introduce problems and actually helps in making WSDL documents self-contained, which is desirable.

4.3. Data model analysis

Correct management of data models is crucial in data centric software systems such as the one under study. As a consequence, we have performed a deeper analysis on this aspect. Table 4 shows metrics that give a rough idea of data type definition, reuse and composition in the three data-sets. The first clear difference is the number of defined data types. The *Direct Migration* data-set contained 182 different data types and 73% of them are defined only once. Since the associated WSDL documents do not share data type definitions, many of the types are duplicated across different WSDL documents. In contrast, 235 unique data types were defined for the WSDL documents of the *Indirect Migration* data-set. Among this set, 104 data types represent business objects, including 39 defined as simple types (mostly enumerations) and 65 defined as complex types, whereas 131 are special XSD constructs needed for making the WSDL documents compliant with the document/wrapped standard structure and hence achieving better interoperability levels with existing service invocation frameworks. This is because, according to the WSDL 1.0 specification, a message part can include either a type or an element attribute to point to its data type. However, according to the WS-I, a message part must only be an element. Therefore, an element was defined for each data type exchanged by input and output messages. For example, the Cuil data type, which represents a business object that is analogous to the Social Security Number in the United States or the National Insurance Number in the United Kingdom, was defined in the shared XSD schema as follows:

```
<xsd:complexType name="Cuil">
    <xsd:sequence>
        <!– Preffix –>
        <xsd:element name="prefijo" type="tns:CuilPrefijo"/>
        <!– Identity document –>
        <xsd:element ref="tns:Documento"/>
        <!– Control (validation) digit –>
        <xsd:element name="digitoControl" type="tns:CuilDigito"/>
    </xsd:sequence>
</xsd:complexType>
```

but also wrapped by an element in the WSDL documents using it as follows:

```
<xsd:element name="cuil" type="ns:Cuil"/>
```

On the contrary, the services from *Direct Migration* define all data types exchanged by their operations as element nodes. Although this practice makes the WSDL documents compliant to the aforementioned existing interoperability standards, it hinders data type definitions reuse, which in turn unnecessarily causes bigger service descriptions that are difficult to be understood. Instead, *Indirect Migration* promotes data type definition reuse. This can be observed in Figure 5, which illustrates the relationships between the *Indirect Migration* Web Services and the 104 data types designed for representing business objects (transitive data type reuse is not illustrated for readability reasons). The diagram was constructed by

Table 4. Data type definition: detailed view.

Data model characteristics/data-set	Direct migration	Indirect migration	Code-first indirect migration
Defined data types	182	235	485
Average definitions per data type	1.83	1.0	3.16
Data types defined only once	133.0 (73%)	235.0 (100%)	126.0 (25%)

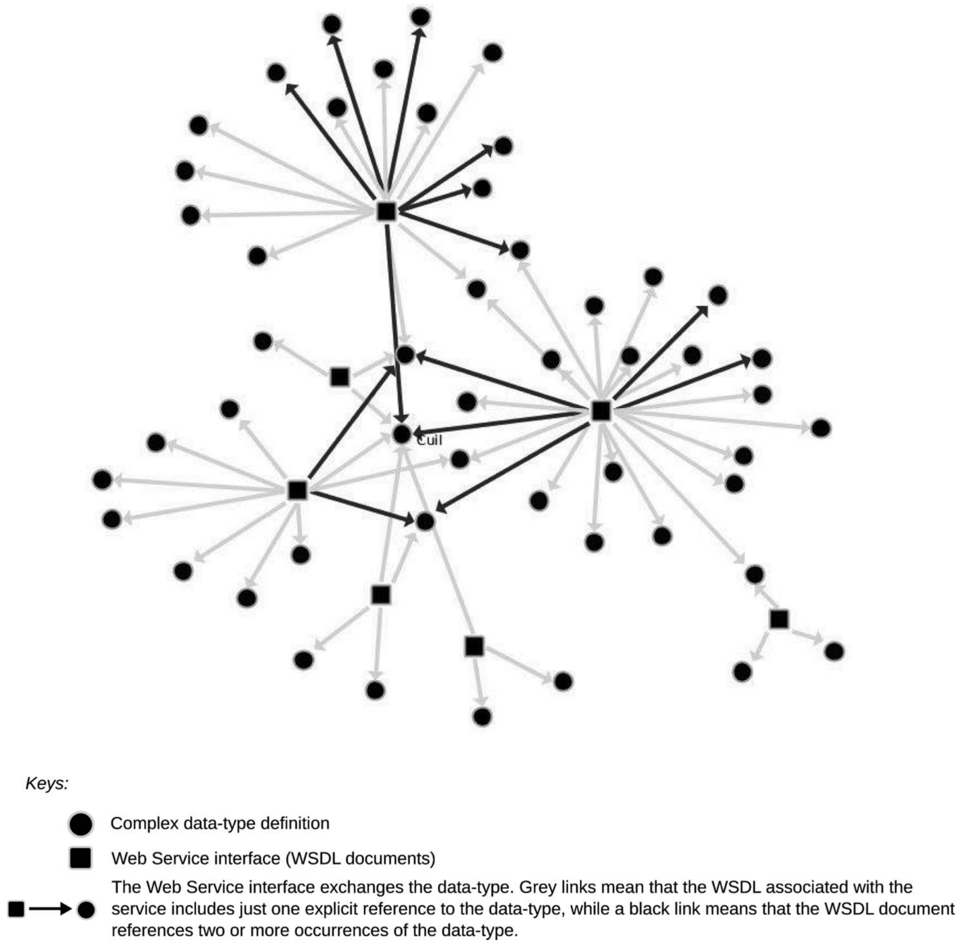
Cuil
Keys:
Complex data-type definition
Web Service interface (WSDL documents)
The Web Service interface exchanges the data-type. Grey links mean that the WSDL associated with the service includes just one explicit reference to the data-type, while a black link means that the WSDL document references two or more occurrences of the data-type.

Figure 5. XSD data type reuse from the refactored Web Services.

using the Guess graph visualisation tool (Adar 2006). A grey link between a service (squares) and an entity or data type (circles) means that the WSDL associated with the former includes just one explicit reference to the latter. Similarly, a black link means that the service interface references two or more entity occurrences. Note that XSD entities with high fan-in are those abstractions that best represent the core business of the organisation. Precisely, one example is the Cuil entity.

Finally, with *Code-First Indirect Migration*, 485 definitions have been obtained and only 25% of these definitions are unique. This low level of data type reuse is due to the inherent problems of automatic WSDL generation. Although the implementation of the Web Services share the various data type definitions, when a WSDL document is generated, the tool cannot determine whether the data type definitions are already included in the XSD of another service. Due to this lack of global knowledge, the tool always map all data types defined by a service, even if this means deriving data type definitions that have been already built.

To sum up, the analysed data-sets of WSDL documents contained 182, 104 and 485 different definitions of business object data types, respectively. The fact that the

WSDL documents of the *Indirect Migration* data-set have fewer data type definitions for representing business objects (104) than the others (i.e. 182 for the *Direct Migration* documents and 485 for the *Code-First Indirect Migration* WSDLs), indicates a better level of data model reuse and a proper utilisation of the XSD complex and element constructors to be WS-I complaint. At the same time, the lack of global knowledge of WSDL generation tools caused that business object definitions belonging to the *Code-First Indirect Migration* data-set quadrupled the definitions within the *Indirect Migration* data-set. This generates bigger WSDL documents, which severely impacts on service discoverability because human discoverers usually prefer smaller WSDL documents over larger ones (Rodriguez *et al*. 2010c).

4.4. Anti-patterns relationships with employed methods and tools

The experiments reported in Section 4.2 show that in the direct migration attempt the WSDL anti-patterns described in the first column of Table 5 were caused by the employed methods or tools of the second column. For instance, the code-first tool employed for generating the ASP Web Service interfaces causes the *Redundant Porty-types* and *Enclosed Data Model* anti-patterns. After investigating the tool, we found that to override its default behaviour it is necessary to extend platform-level services to include a custom C# to WSDL converter, which might not be feasible for many migration processes.

At the same time, the occurrences of the *Inappropriate or lacking comments* anti-pattern within the *Direct Migration* data-set were not caused by the code-first tool. Instead, they were caused because developers did not employ this tool properly. To include documentation comments in target WSDL documents using .NET ASP framework services, and thus avoiding the mentioned anti-pattern, developers should add the [Description='comments'] tag in the source code of the main service implementation class during the third step of the migration method of Section 3.2.

We have also found that many of the occurrences of the *Ambiguous Name* anti-pattern in the WSDL documents of the direct migration attempt could have been avoided if the IT department would had manually improved parameter names of generated COM+ objects or C# wrappers during steps 1 and 2, respectively. Clearly, this represents a trade-off between automatism or migration speed and name quality.

Table 5. Relationships between anti-patterns and methods/tools of the first migration attempt.

Anti-patterns	Method/tool-set	Alternative
Inappropriate or lacking comments	Step 3	Add the [Description = 'comments'] tag in the source code of the main service implementation class during this step
Ambiguous names	Steps 1 and 2	Improve parameter names of generated COM + objects or C# wrappers during these two steps
Redundant port-types; enclosed data model	ASP WSDL generator	Extend platform-level services to include a custom WSDL generator

5. Conclusions and future work

Moving from mainframe systems to service-oriented environments is not only a complex task, but also a sensitive one because such systems usually form the backbone of large financial organisations and governments. There are many bibliographical references that support the argument that the direct migration approach, materialised through wrapping techniques, enables the effective modernisation of legacy systems to SOA. Bringing the migration of a large system into fruition with the indirect migration approach, on the other hand, it is widely recognised as being harder compared to the direct migration approach.

The pros and cons of either migration approaches is something very well known, discussed and undoubted in the software industry. Many case studies reporting either direct or indirect migration experiences exist in the literature. However, as far as we know there is not a study comparing the outcomes of employing both approaches on the same real world large enterprise system, mainly because managing two migration attempts is too costly for any enterprise. Instead, in this paper, we describe a large real COBOL system that has been migrated to Web Services by using both approaches. This paper analysed the outcome of applying both approaches for migrating and building services from a real system.

Table 6 presents an illustrative comparison of the resources needed by each migration attempt. The first migration attempt succeeded in delivering Web Services within a short period of time and without expending lots of resources, by employing a direct migration approach with wrapping. As shown in the second column of the table, with the associated methods and tool-set, it only took five days and one developer to migrate 32 CICS/COBOL transactions. It is worth noting that the first attempt was costless since no software licenses had to be bought, and no developers had to be hired, i.e. regular members of the IT department performed the first migration attempt. However, this could be not the case for many enterprises and therefore there may be costs associated to buying the necessary tool-set and hiring external manpower when performing a direct migration with wrapping. In contrast, an indirect migration attempt with re-engineering was much more expensive and required more time to be completed. In particular, eight developers, three Web Services specialists, 13 months and 320,000 US dollars for re-engineering the same 32 transactions were required. For this attempt, external specialists and developers were hired, whose salaries have been included in this cost.

Although there were differences in the resources demanded by each attempt, it is important to note that the second attempt provided better services (in terms of interface clarity) and removed the CICS/COBOL software from the system. Table 7 summarises these findings. This latter means that the system business logic is now

Table 6. Costs comparison of the 32 prioritised transactions migration.

Resources	First attempt: direct migration	Second attempt: indirect migration
Developers	1	8
Specialists	0	3
Time	Five days	13 months
Money	US$ 0[a]	US$ 320,000

Note: [a]The agency owns required software licenses and one of its regular employees performed the migration attempt.

Table 7. Qualitative comparison of the achieved SOA systems.

Migration approach	System implementation			WSDL interface design		
	Target platform	Maintainability	Mainframe independence	Discoverability	Reusability	Understandability
Direct	COBOL + .NET	Harder	No	Poor	Poor	Regular
Indirect	.NET	Easier	Yes	Very good	Very good	Very good

written in a more modern platform, using a modern programming paradigm as well, and for which it is easier and cheaper to acquire productive development tools or hire developers. Furthermore, the mainframe may be made cease operation. Moreover, though the second attempt was more expensive, it may be indirectly helping to reduce other costs, namely, payments for processing power and expensive salaries to hire savvy programmers that master the old technologies. Indeed, by removing the prioritised transactions' load from the mainframe (only six of them represented the 56% of total mainframe load) the agency saves the money invested in the second migration, because the IT department estimated that due to the mainframe growing workload the agency would had to rent more processing power by the end of the same semester at a cost of 2 millions US dollars a year.

At the same time, achieved results show that the Web Services obtained with the indirect migration present better quality attributes in respect to two perspectives, namely, interface understandability and reusability. In this sense, interfaces are more concise, have more representative names for their port-types, operations and messages, have proper comments and allow for better levels of granularity and functional cohesion. The definitions of business objects present improvements as well, since fewer objects are defined while there are no repeated definitions. With regard to the catalogue of WSDL discoverability anti-patterns of Rodriguez *et al.* (2010c), the service interfaces generated by following the indirect migration and contract-first combination do not present such undesirable anti-patterns. All in all, better WSDL interfaces positively impacts on the development of client applications, by reducing client applications developers' effort needed to discover and understand available services (Rodriguez *et al.* 2010c).

Due to the positive results achieved in the second attempt of the project, a continuation of this attempt is under negotiation. Specifically, the remaining services that were also migrated during the first attempt of the project will possibly be migrated following the indirect migration approach and contract-first techniques to build Web Services, which in turn will be benefited by service-level agreements (SLA) that define the behaviour and quality of services non-functional properties (Unger *et al.* 2009). This will provide us with more input data in the form of larger WSDL and XSD data-sets so as to provide more empirical evidence and thus further support our claims. On the other hand, another line of future research includes the development of software tools for automatically identifying and applying refactoring opportunities in legacy systems. In this sense, we are working on heuristics for detecting data model design problems in directly migrated CICS/COBOL transactions and programs, and grouping different transactions based on their functional cohesion degrees. For assessing functional cohesion, we are defining an heuristic that bases on an existing algorithm to detect services having low cohesive operations (Rodriguez *et al.* 2010b). To validate the effectiveness of our approach, we will compare the output of the resulting tools with the manual results obtained via the indirect migration results reported in this paper. We could eventually also employ the output of the third phase of the project.

Notes

1. http://www.gartner.com
2. SLOC metric for COBOL source code was calculated using the SLOC Count utility available at http://www.dwheeler.com/sloccount, which does not count commented lines.
3. Basic Profile Version 1.1: http://www.ws-i.org/Profiles/BasicProfile-1.1.html

4. Professional and paid versions of the two first tools were used, whereas a standard and free version of soap UI was employed.
5. MyBatis, http://www.mybatis.org/dotnet.html
6. IBM's DB2Connect, http://www-01.ibm.com/software/data/db2/db2connect/

References

Adar, E., 2006. GUESS: A language and interface for graph exploration. *In*: *Conference on human factors in computing systems*, 22–27 April, Montreal, Quebec, Canada. New York: ACM Press, 791–800.

Almonaies, A., Cordy, J., and Dean, T., 2010. Legacy system evolution towards service-oriented architecture. *In*: *International workshop on SOA Migration and Evolution (SOME)*, 15 March, Madrid, Spain OFFIS e. V. Escherweg 2 26121, Oldenburg, Germany, 53–62.

Beaton, J., *et al*., 2008. Usability challenges for enterprise service-oriented architecture APIs. *In*: *IEEE symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 15–19 September, Herrsching am Ammersee, Germany. Los Alamitos, CA: IEEE Computer Society, 193–196.

Bichler, M. and Lin, K.J., 2006. Service-oriented computing. *Computer*, 39 (3), 99–101.

Blake, M.B. and Nowlan, M.F., 2008. Taming Web Services from the wild. *Internet Computing*, 12 (5), 62–69.

Colosimo, M., *et al*., 2009. Evaluating legacy system migration technologies through empirical studies. *Information and Software Technology*, 51 (2), 433–447.

Crasso, M., *et al*., 2010. Revising WSDL documents: why and how. *Internet Computing*, 14 (5), 30–38.

De Lucia, A., *et al*., 2008. Developing legacy system migration methods and tools for technology transfer. *Software Practice Experience*, 38 (13), 1333–1364.

D'Mello, D.A. and Ananthanarayana, V.S., 2010. Dynamic selection mechanism for quality of service aware Web Services. *Enterprise Information Systems*, 4, 23–60.

Erickson, J. and Siau, K., 2008. Web service, service-oriented computing, and service-oriented architecture: separating hype from reality. *Journal of Database Management*, 19 (3), 42–54.

Erl, T., 2007. *SOA principles of service design*. Upper Saddle River, New Jersey, USA: Prentice-Hall.

Fan, J. and Kambhampati, S., 2005. A snapshot of public web services. *SIGMOD Record*, 34 (1), 24–32.

Gao, S.S., *et al*., 2009. Technical report, W3C Consortium. Available from: http://www.w3.org/TR/xmlschema11-1 [Accessed 7 August 2012].

Gong, Z., Muyeba, M., and Guo, J., 2010. Business information query expansion through semantic network. *Enterprise Information Systems*, 4, 1–22.

Gudgin, M., *et al*., 2007. Technical report, W3C Consortium. Available from: http://www.w3.org/TR/soap12-part1 [Accessed 7 August 2012].

Kohlborn, T., *et al*., 2009. Identification and analysis of business and software services – a consolidated approach. *IEEE Transactions on Services Computing*, 2 (1), 50–64.

Leinecker, R.C., 2000. *Com+Unleashed*. Indianapolis, Indiana, USA: Sams.

Li, S.H., *et al*., 2007. Migrating legacy information systems to web services architecture. *Journal of Database Management*, 18 (4), 1–25.

Mateos, C., *et al*., 2010. Separation of concerns in service-oriented applications based on pervasive design patterns. *In*: *Web Technology Track (WT) – 25th ACM symposium on applied computing (SAC '10)*, 22–26 March, Sierre, Switzerland. New York: ACM Press, 2509–2513.

Mietzner, R., Leymann, F., and Unger, T., 2011. Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications. *Enterprise Information Systems*, 5, 59–77.

Papazoglou, M. and van den Heuvel, W.J., 2006. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2 (4), 412–442.

Ricca, F. and Marchetto, A., 2009. A "quick and dirty" meet-in-the-middle approach for migrating to SOA. *In*: *Proceedings of the joint international and annual ERCIM workshops on principles of software evolution (IWPSE) and software evolution (Evol) workshops (IWPSE-Evol '09)*, 24–28 August, Amsterdam, The Netherlands. New York, NY, USA: ACM, 73–78.

Rodriguez, J.M., *et al.*, 2010a. The EasySOC project: a rich catalog of best practices for developing web service applications. *In*: *Jornadas Chilenas de Computación (JCC) – INFONOR 2010, Antofagasta, Chile* SCC (Sociedad Chilena de la Ciencia de la Computación), 33–42.

Rodriguez, J.M., *et al.*, 2010b. Automatically detecting opportunities for web service descriptions improvement. *In*: *10th IFIP WG 6.11 conference on e-Business, e-Services, and e-Society (I3E 2010), Ciudad Autónoma de Buenos Aires*, 3–5 November, Argentina, Vol. 431. Boston: Springer, 139–150.

Rodriguez, J.M., *et al.*, 2010c. Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75 (11), 1001–1021.

Unger, T., Mietzner, R., and Leymann, F., 2009. Customer-defined service level agreements for composite applications. *Enterprise Information Systems*, 3 (3), 369–391.

Wang, K., *et al.*, 2010. A service-based framework for pharmacogenomics data integration. *Enterprise Information Systems*, 4, 225–245.

Yourdon, E. and Constantine, L.L., 1979. *Structured design: fundamentals of a discipline of computer program and systems design*. Upper Saddle River, NJ, USA: Prentice-Hall.

Appendix 1. Transaction migration example

The next code shows a simple COBOL transaction, which exemplifies the different artefacts that were performed and generated, respectively, when migrating an individual transaction with the strategy and technologies described in Section 3.2. The example transaction just displays a prompt to the user, blocks until data is manually entered and then displays the entered message. For simplicity, the transaction does not contain extra programs and the COMMAREA has been defined in an in-line way (lines 5–8).

```
000001    ID DIVISION.
000002    PROGRAM-ID SAMPLE.
000003    DATA DIVISION.
000004    WORKING STORAGE SECTION.
000005    01 OUT-MSG.
000006        02 FILLER PIC X(20) VALUE "YOU HAVE ENTERED: ".
000007        02 MSG PIC X(20).
000008    01 INP-MSG PIC X(20) VALUE "ENTER A MESSAGE: ".
000009    PROCEDURE DIVISION.
000010    DISPLAY.
000011    EXEC CICS SEND FROM(INP-MSG) ERASE END-EXEC.
000012    EXEC CICS RECEIVE INTO(MSG) END-EXEC.
000013    EXEC CICS SEND FROM(OUT-MSG) ERASE END-EXEC.
000014    EXEC CICS RETURN END-EXEC.
```

Here, INP-MSG and OUT-MSG specify the data type structure and length of the COMMAREA, but at the same time define the 'interface' of the transaction to the outer world. Then, a terminal application executing this transaction may obtain a screen output such as:

```
ENTER A MESSAGE:
HELLO, WORLD!
YOU HAVE ENTERED: HELLO, WORLD!
```

At this point, the example consists of an entire COBOL program containing a COMMAREA that structurally defines the program interface. The COMTI Component Builder tool receives a COMMAREA as input to automatically derive a type library (TLB).

The TLB is accessible from any component of the .NET framework as a COM+ object afterwards. The developer is asked to name the new type and its operations.

VisualStudio 2008 provides built-support for instantiating COM+ objects and calling their operations. This support automatically builds a local C# object that wraps a (possibly remote) COM+ object. Finally, specific source code annotations were included in the header of each C# class and its operations to enable the automatic generation of WSDL documents at the .NET framework level. This is because the .NET framework requires developers to explicitly indicate what to expose as the service interface and which methods to include as service operations within it.

Then, the new type is packaged in a DLL library and a TLB is generated as follows:

```
**************************************************************
*          METHOD NAME.................HELLO_WORLD
*          TRANSACTION PROGRAM NAME.....SAMPLE
**************************************************************
01   HELLO_WORLD-INPUT-AREA.
       05   INP-MSG      PIC X(20).     INPUT
01   HELLO_WORLD-OUTPUT-AREA.
       03   FILLER
              04      PIC X(20).      OUTPUT
       03   MSG PIC X(20).        OUTPUT
* BYTES THIS HOST PROGRAM SENDS......40
* BYTES THIS HOST PROGRAM RECEIVES...20
**************************************************************
```

To call the generated type from a C# class, the former should be wrapped as a COM+ object. Once a COM+ object has been created, VisualStudio 2008 allows for calling the COM+ object through the C# class named C_SAMPLE. Finally, to expose C_SAMPLE as a Web Service, another class (in this case, ExampleWebService) must be created, which is annotated with specific meta-data that instructs the .NET framework about how to map the methods of the class to Web Service operations. Below, the complete source code of such a class is shown.

```
[WebService]
public class ExampleWebService {
    [WebMethod]
    public string Hello_Word(string inp-msg) {
        C_SAMPLE comPlusObject=null;
        comPlusObject=Server.CreateObject("SAMPLE");
        string r=null;
        r=comPlusObject.HELLO_WORLD(inp-msg).MSG;
        return "YOU HAVE ENTERED: " + r;
    }
}
```

As the reader can see, before calling the COM+ object (line 8), an instance of it is created (line 6). Moreover, lines 1 and 3 show the annotations required by the .NET framework for automatically generating the WSDL document:

```
<wsdl:definitions>
  <wsdl:types>
    <s:schema elementFormDefault="qualified">
      <s:element name="HelloWorld">
        <s:complexType>
            <s:sequence>
              <s:element minOccurs="0" maxOccurs="1"
                name="inp_msg" type="s:string"/>
        </s:sequence>
        </s:complexType>
      </s:element>
```

```
            < s:element name="HelloWorldResponse" >
               < s:complexType >
                  < s:sequence >
                     < s:element minOccurs="0" maxOccurs="1"
                       name="HelloWorldResult" type="s:string" />
                     < /s:sequence >
               < /s:complexType >
            < /s:element >
         < /s:schema >
   < /wsdl:types >
   < wsdl:message name="HelloWorldSoapIn" >
            < wsdl:part name="parameters" element="tns:HelloWorld" />
   < /wsdl:message >
   < wsdl:message name="HelloWorldSoapOut" >
            < wsdl:part name="parameters" element="tns:HelloWorldResponse" />
   < /wsdl:message >
   < wsdl:portType name="ExampleWebServiceSoap" >
            < wsdl:operation name="HelloWorld" >
               < wsdl:input message="tns:HelloWorldSoapIn" />
               < wsdl:output message="tns:HelloWorldSoapOut" />
         < /wsdl:operation >
      < /wsdl:portType >
   < /wsdl:definitions >
```