This is a preprint of the article: "G. Salvatierra, M. Crasso, C. Mateos and A. Zunino: "Towards a Computer Assisted Approach for Migrating Legacy Systems to SOA". Lecture Notes in Computer Science (12th International Conference on Computational Science and Its Applications - ICCSA 2012). Vol. 7336, pp. 484-497. Springer-Verlag. 2012. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_36

Towards a computer assisted approach for migrating legacy systems to SOA

Gonzalo Salvatierra[2], Cristian Mateos[1,2,3], Marco Crasso[1,2,3], and Alejandro Zunino[1,2,3]

[1] ISISTAN Research Institute.
[2] UNICEN University.
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

Abstract. Legacy system migration to Service-oriented Architectures (SOA) has been identified as the right path to the modernization of enterprise solutions needing agility to respond to changes and high levels of interoperability. However, one of the main challenges of migrating to SOA is finding an appropriate balance between migration effort and the quality of resulting service interfaces. This paper describes an approach to assist software analysts in the definition of produced services, which bases on the fact that poorly designed service interfaces may be due to bad design and implementation decisions present in the legacy system. Besides automatically detecting common design pitfalls, the approach suggests refactorings to correct them. Resulting services have been compared with those that resulted from migrating a real system by following two classic approaches.

Keywords: SERVICES-ORIENTED ARCHITECTURE, WEB SERVICES, LEGACY SYSTEM MIGRATION, DIRECT MIGRATION, INDIRECT MIGRATION, SEMI-AUTOMATIC COBOL MIGRATION

1 Introduction

From an operational standpoint, many enterprises and organizations rely on out-dated systems developed using old languages such as COBOL. These kind of systems are known as *legacy systems*. Still, enterprises have to face high costs for maintaining their legacy systems mainly because three factors [1]. First, these systems usually run on (expensive) mainframes that must be rented. Second, it is necessary to continuously hire and retain developers specialized in legacy technologies, which is both expensive and rather difficult. Third, in time these old systems have suffered modifications or upgrades for satisfying variable business goals. For example, most banks nowadays offer Home Banking services, though most bank systems were originally written in COBOL. Therefore, it is common to find a pre-Web, 50 year old technology working alongside modern platforms (e.g. JEE or .Net) within the same information system.

In this sense, migration becomes a necessity. Currently, the commonest target for migrating legacy systems is SOA (Service-Oriented Architecture) [2], by which systems are composed of pieces of reusable functionalities called *services*. Services are usually materialized through independent server-side "components" –i.e. Web Services [3]–

that are exposed via ubiquitous Web protocols. Once built, Web Services can be remotely composed by heterogeneous client applications, called *consumers*.

Recent literature [4] identifies two broad approaches for migrating a legacy system: *direct migration* and *indirect migration*. The former consists of simply wrapping legacy system programs with Web Services. This practice is cheap and fast, but it does not succeed in completely replacing the legacy system by a new one. On the other hand, indirect migration proposes completely re-implementing a legacy system using a modern platform. This is intuitively expensive and time consuming because not only the system should be reimplemented and re-tested, but also the business logic should be reverse-engineered because system documentation could have been lost or not kept up-to-date.

In SOA terms, an important difference between direct migration and indirect migration is the quality of the *SOA frontier*, or the set of WSDL documents exposed to potential consumers after migration. Web Service Description Language (WSDL) is an XML standard for describing a service interface as a set of operations with input and output data-types. Although service interfaces quality is a very important factor for the success of a SOA system [5,6,7], most enterprises use direct migration because of its inherent low cost and shorter time-to-market, but derived WSDLs are often a mere low-quality, Web-enabled representation of the legacy system program interfaces. Then, services are not designed with SOA best design practices, which ensure more reusable interfaces. On the other hand, indirect migration provides the opportunity to introduce improvements into the legacy logic (e.g., unused parameters and duplicate code elimination) upon migrating the system, therefore improving the SOA frontier.

In this paper, we propose an approach called *assisted migration* that aims at semi-automatically obtaining SOA frontiers with similar quality levels to that of indirect migration but by reducing its costs. The approach takes as input a legacy system migrated using direct migration, which is a common scenario, and performs an analysis of possible refactoring actions to increase the SOA frontier quality. Although assisted migration does not remove the legacy system, the obtained SOA frontier can be used as a starting point for re-implementing it. This allows for a smoother system replacement because the (still legacy) implementation can be replaced with no harm to consumers since the defined service interfaces remain unmodified.

To evaluate our approach, we used two real SOA frontiers obtained by directly as well as indirectly migrating the same legacy COBOL system [1], which is owned by a large Argentinean government agency. We applied the assisted migration approach by feeding it with the direct migration version of the original system. After that, we compared the three obtained SOA frontiers in terms of cost, time, and quality. The results show that our approach produces a SOA frontier nearly as good as that the indirect migration, but at a cost similar to that of direct migration.

2 Automatic detection of SOA frontier improvement opportunities

We have explored the hypothesis that enhancing the SOA frontier of a migrated system can be done in a fast and cheap manner by automatically analyzing its legacy source code, and supplying software analysts with guidelines for manually refining the WSDL documents of the SOA frontier based on the bad smells present in the source code. This

is because many of the design problems that occur in migrated service interfaces may be due to design and implementation problems of the legacy system.

The input of the proposed approach is a legacy system source code and the SOA frontier that result from its direct migration to SOA, concretely the CICS/COBOL files and associated WSDL documents. Then, this approach can be iteratively executed for generating a new SOA frontier at each iteration. The main idea is to iteratively improve the service interfaces by removing those *WSDL anti-patterns* present in them.

A WSDL anti-pattern is a recurrent practice that prevents Web Services from being discovered and understood by third-parties. In [8] the authors present a catalog of WSDL anti-patterns and define each of them way by including a description of the underlying problem, its solution, and an illustrative example. Counting the anti-patterns occurrences within a WSDL-based SOA frontier then gives a quantitative idea of frontier quality, because the fewer the occurrences are, the better the WSDL documents are in terms of reusability, thus removing WSDL anti-patterns root causes represents SOA frontier improvement opportunities. Note that this kind of assessment also represents a mean to compare various SOA frontiers obtained from the same system.

The proposed approach starts by automatically detecting potential WSDL anti-patterns root causes within the migrated system given as input ("Anti-patterns root causes detection" activity). Then, the second activity generates a course of actions to improve the services frontier based on the root causes found ("OO refactorings suggestion" activity). The third activity ("OO refactorings application") takes place when software analysts apply all or some of the suggested refactoring actions. Accordingly, at each iteration a new SOA frontier is obtained, which feds back the first activity to refine the anti-patterns root causes detection analysis. Figure 1 depicts the proposed approach.

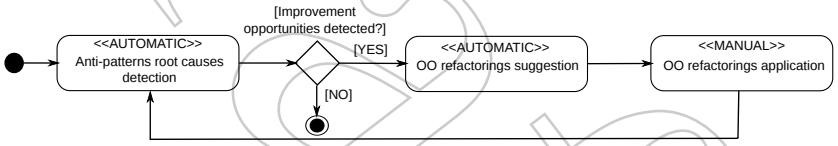

Fig. 1. Software-assisted SOA frontier definition activities.

2.1 WSDL anti-patterns root causes detection

The anti-pattern root causes detection activity is performed automatically, since manually revising legacy system source code is a cumbersome endeavor. To do this, we have defined and implemented the ten heuristics summarized in Table 1. A defined heuristic receives the implementation or the WSDL document of a migrated transaction and outputs whether specific evidence of anti-patterns root causes is found.

We based on the work of [7] for designing most of the heuristics proposed in this paper, since [7] presents heuristics for automatically detecting the anti-patterns in a given WSDL document. Additionally, we have specially conceived heuristics 7 and 8

Table 1. Heuristics for detecting evidence of WSDL anti-patterns root causes.

Id and description	Input	True when
1. Look for comments in WSDL <documentation> elements	WSDL document	At least one operation lacks documentation.
2. Search inappropriate names for service elements	WSDL document	The length of a name token is lower than 3 characters, or when token refers to a technology, or when an operation name contains two or more verbs or an argument name contains a verb.
3. Detect operations that receive or return too many parameters	WSDL document	At least one operation input/output has more than P parameters.
4. Look for error information being exchanged as output data	WSDL document	An output message part has any of the tokens: “error”, ”errors”, ”fault”, ”faults”, ”fail”, ”fails”, ”exception, ”exceptions, ”overflow”, ”mistake”, ”misplay”.
5. Look for redundant data-type definitions	WSDL document	At least two XSD data-types are syntacticly identical.
6. Look for data-types with inconsistent names and types	WSDL document	The name of a parameter denotes a quantity but it is not associated with a numerical data-type (e.g. numberOfChildren:String).
7. Detect not used parameters	COBOL code	At least one parameter is not associated with a COBOL MOVE statement.
8. Look for shared dependencies among services implementations	COBOL code	The list of COBOL programs that are copied, or included, or called from two or more service implementations is not empty.
9. Look for data-types that subsumes other data-types	WSDL document	An XSD complex data-type contains another complex XSD data-type, or a list of parameters subsumes another list of parameters.
10. Detect semantically similar services and operations	WSDL document	A vectorial representation of the names and associated documentation of two services or operations, are near in a vector space model.

for analyzing COBOL source, while heuristics 6 and 9 for supporting two relevant anti-patterns that had not been identified in [7]. Concretely, the heuristic 6 analyzes names and data-types of service operations parameters to look for known relationships between names and types. Given a parameter, the heuristic splits the parameter name by basing on classic programmers’ naming conventions, such as Camel Casing and Hungarian notation. Each name token is compared to a list of keywords with which a data-type is commonly associated. For example, the token “birthday” is commonly associated with the XSD built-in xsd:date data-type, but the token “number” with the xsd:int data-type. Therefore, the heuristic in turn checks whether at least one name token is wrongly associated with the data-type.

The heuristic 7 receives the COBOL code of a migrated program and checks whether every parameter of the program output COMMAREA is associated with the COBOL MOVE assignation statement. In other words, given a COBOL program, the heuristic retrieves its output COMMAREA, then gets every parameter from within it (including parameters grouped by COBOL records), and finally looks for MOVE statements having the declared parameter. One temporal limitation of this heuristic is that the search for MOVE statements is only performed in the main COBOL program, whereas copied

or included programs are left aside, and those parameters that are assigned by the execution of an SQL statement are ignored by this heuristic.

The heuristic 8 receives two COBOL programs as input. Then, for both programs it separately builds a list of external COBOL programs, copies, and includes, which are called (it looks for the CALL reserved word) from the main program, and finally checks whether the intersection of both lists is empty or not.

The heuristic 9 receives a WSDL document as input and detects the inclusion of one or more parameters of a service operation in the operations of another service. To do this, parameter names and data-types are compared. For comparing names classic text preprocessing techniques are applied, namely split combined words, remove stopwords, and reduce names to stems. For comparing data-types the heuristic employs the algorithm named *Redundant Data Model*, which is presented in [7].

Once we have all the evidence gathered by the heuristics, the mere presence of anti-pattern root causes represent opportunities to improve a SOA frontier. Formally, $id \rightarrow opportunity_i$ means that $opportunity_i$ may be present when the heuristic *id* detects its associated root causes (i.e. outputs 'true'):

$$4 \rightarrow Improve\ error\ handling\ definitions$$
$$8 \rightarrow Expose\ shared\ programs\ as\ services$$
$$10 \rightarrow Improve\ service\ operations\ cohesion$$

The first rule is for relating the opportunity to split an output message in two, one for the output data and another for fault data, when there is evidence of the output message conveying error data (heuristic 4). The second rule unveils the opportunity to expose shared programs as Web Services operations, when a COBOL program is called from many other programs, i.e. its fan-in is higher than a threshold T (heuristic 8). The third rule associates the opportunity to improve the cohesion of a service with evidence showing low cohesion among service operations (heuristic 10).

In the cases shown below, however, the evidence gathered by an heuristic does not support improvement opportunities by itself, and two heuristics must output 'true':

$$8\ and\ 9 \rightarrow Remove\ redundant\ operations$$
$$1\ and\ 2 \rightarrow Improve\ names\ and\ comments$$

The first rule is for detecting the opportunity to remove redundant operations, and is fired if two or more COBOL programs share the same dependencies (heuristic 8) but also the parameters of one program subsume the parameters of the other program (heuristic 9). The rationale behind this rule is that when two service operations not only call the same programs, but also expose the same data as output –irrespective of the amount of exposed data– there is an opportunity to abstract these operations in another unique operation. The second rule indicates that there is the opportunity to improve the descriptiveness of a service operation when it lacks documentation (heuristic 1) and its name or the names of its parameters are inappropriate (heuristic 2).

Finally, the rule $3\ or\ 5\ or\ 6\ or\ 7 \rightarrow Improve\ business\ object\ definitions$ combines more than one evidence for readability purposes, and it is concerned with detecting an opportunity to improve the design of the operation in/out data-types. The opportunity is suggested when the operation exchanges too many parameters (heuristic 3), there are

This is a preprint of the article: "G. Salvatierra, M. Crasso, C. Mateos and A. Zunino: "Towards a Computer Assisted Approach for Migrating Legacy Systems to SOA". Lecture Notes in Computer Science (12th International Conference on Computational Science and Its Applications - ICCSA 2012). Vol. 7336, pp. 484-497. Springer-Verlag. 2012. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_36

repeated data-type definitions (heuristic 5), the type of a parameter is inconsistent with its name (heuristic 6), or there are unused parameters (heuristic 7).

2.2 Supplying guidelines to improve the SOA frontier

The second activity of the proposed approach consists of providing practical guidelines to apply detected improvement possibilities. These guidelines consist of a sequence of steps that should be revised and potentially applied by software analysts. The proposed guidelines are not meant to be automatic, since there is not a unique approach to build or modify a WSDL document and, in turn, a SOA frontier [9].

The cornerstone of the proposed guidelines is that classic Object-Oriented (OO) refactorings can be employed to remove anti-patterns root causes from a SOA frontier. This stems from the fact that services are described as OO interfaces exchanging messages, whereas operation data-types are described using XSD, which provides some operators for expressing encapsulation and inheritance. Then, we have organized a sub-set of Fowler et al.'s catalog of OO refactorings [10], to provide a sequence of refactorings that should be performed for removing each anti-pattern root cause.

Table 2. Association between SOA frontier refactorings and Fowler et al.'s refactorings.

SOA Frontier Refactoring	**Object-Oriented Refactoring**
Remove redundant operations	1: Extract Method **or** Extract Class
Improve error handling definition	1: Replace Error Code With Exception
Improve business objects definition	1: Convert Procedural Design to Object **and** Replace Conditional with Polymorphism 2: Inline Class 3: Extract Class **and** Extract Subclass **and** Extract Superclass **and** Collapse Hierarchy 4: Remove Control Flag **and** Remove Parameter 5: Replace Type Code with Class **and** Replace Type Code with Subclasses
Expose shared programs as services	1: Extract Method **or** Extract Class
Improve names and comments	1: Rename Method **or** Preserve Whole Object **or** Introduce Parameter Object **or** Replace Parameter with Explicit Methods
Improve service operations cohesion	1: Inline Class **and** Rename Method 2: Move Method **or** Move Class

The proposed guidelines associate a SOA frontier improvement opportunity (Table 2, first column) with one or more OO refactorings, which are arranged in sequences of optional or mandatory refactoring combinations (Table 2, second column). Moreover, for "*Improve business objects definition*" and "*Improve service operations cohesion*", the associated refactorings comprise more than one step. Hence, at each individual step analysts should apply the associated refactorings combinations as explained.

Regarding how to apply OO refactorings, it depends on how the WSDL documents of the SOA frontier have been built. Broadly, there are two main approaches to build

WSDL documents, namely *code-first* and *contract-first* [9]. Code-first refers to automatically extract service interfaces from their underlying implementation. On the other hand, with the contract-first approach, developers should first define service interfaces using WSDL, and supplying them with implementations afterwards. Then, when the WSDL documents of a SOA frontier have been implemented under code-first, the proposed guidelines should be applied on the outermost components of the services implementation. Instead, when contract-first has been followed, the proposed OO refactorings should be applied directly on the WSDL documents.

For instance, to remove redundant operations from a code-first Web Service, developers should apply the "Extract Method" or the "Extract Class" refactorings on the underlying class that implements the service. In case of a contract-first Web Service, by extracting an operation or a port-type from the WSDL document of the service, developers apply the "Extract Method" or the "Extract Class" refactorings, but developers should also update service implementations for each modified WSDL document.

To sum up, in this section we presented an approach to automatically suggest how to improve the SOA frontier of a migrated system. This approach has been designed for reducing the costs of supporting an indirect migration attempt, while achieving a better SOA frontier than with a direct migration one. In this sense, the next section provides empirical evidence on the SOA frontier quality achieved by modernizing a legacy system using the direct, indirect, and this proposed approach to migration.

3 Evaluation

We have compared the service frontier quality achieved by direct and indirect (i.e. manual approaches), and our software-assisted migration (semi-automatic) by migrating a portion of a real-life COBOL system comprising 32 programs (261,688 lines of source code) accessing a database of around 0.8 Petabytes. Three different service frontiers were obtained: "Direct Migration", "Indirect Migration", and "Assisted Migration". The comparison methodology consisted of analyzing:

- *Classical metrics*: We employed traditional software metrics, i.e. lines of code (LOC), lines of comments, and number of offered operations in the service frontiers. In this context higher values means bigger WSDL documents, which compromises clarity and thus consuming services is more difficult to application developers.
- *Data model*: Data model management is crucial in *data-centric* software systems such as the one under study. We analyzed the data-types produced by each migration approach to get an overview of data-type quality within the service frontiers. For example, we have analyzed business object definitions reuse by counting repeated data-type definitions across WSDL documents.
- *Anti-patterns*: We used the set of service interface metrics for measuring WSDL *anti-patterns* occurrences described in WSDL [8]. We used anti-patterns occurrences as an inverse quality indicator (i.e. fewer occurrences means better WSDLs).
- *Required Effort*: We used classic time and human resources indicators.

Others non-functional requirements, such as performance, reliability or scalability, have not been considered since we were interested in WSDL document quality only.

This is a preprint of the article: "G. Salvatierra, M. Crasso, C. Mateos and A. Zunino: "Towards a Computer Assisted Approach for Migrating Legacy Systems to SOA". Lecture Notes in Computer Science (12th International Conference on Computational Science and Its Applications - ICCSA 2012). Vol. 7336, pp. 484-497. Springer-Verlag. 2012. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_36

3.1 Classical metrics analysis

Table 3. Classical metrics and data model analysis: Obtained results.

Frontier	# of files		Total operations	Average LOC	
	WSDL	XSD		Per file	Per operation
Direct migration	32	0	39	157	129
Indirect migration	7	1	45	495	88
Assisted migration	16	1	41	235	97

Data-set	Defined data-types	Definitions per data-type (less is better)	Unique data-types (more is better)
Direct Migration	182	1.29 (182/141)	141 (73%)
Indirect Migration	235	1.00 (235/235)	235 (100%)
Assisted Migration	191	1.13 (191/169)	169 (88%)

As Table 3 (top) shows, the Direct Migration data-set comprised 32 WSDL documents, i.e. one WSDL document per migrated program. Contrarily, the Indirect Migration and Assisted Migration data-sets had 7 WSDL documents and 16 WSDL documents, respectively. Having less WSDL documents means that several operations were grouped in the same WSDL document, which improves cohesion since these WSDL documents were designed to define functional related operations. Another advantage observed in the Indirect Migration and Assisted Migration data-sets over the Direct Migration data-set was the existence of an XSD file for sharing common data-types.

Secondly, the number of offered operations was 39, 45, and 41 for Direct Migration, Indirect Migration, and Assisted Migration frontiers. Although there were 32 programs to migrate, the first frontier had 39 operations because one specific program was divided into 8 operations. This program used a large registry of possible search parameters plus a control couple to select which parameters to use upon a particular search. After migration, this program was further wrapped with 6 operations with more descriptive names each calling the same COBOL program with a different control couple.

The second and third frontiers generated even more operations for the same 32 programs. This was mainly caused as *functionality disaggregation* and *routine servification* were performed during frontier generation. The former involves mapping COBOL programs that returned too many output parameters with various purposes to several purpose-specific service operations. Furthermore, the latter refers to exposing as services “utility” programs called by many other programs. Then, what used to be COBOL internal routines also become part of the SOA frontier.

Finally, although the Indirect Migration frontier had the highest LOC per file, it also had the lowest LOC per operation. The Assisted Migration frontier had a slightly higher LOC per operation than the Indirect Migration frontier. In contrast, the LOC per operation of the Direct Migration frontier was twice as much as that of the other two frontiers. Then, more code has to be read by consumers in order to understand what an operation does and how to call it, and hence WSDL documents are more cryptic.

3.2 Data model analysis

Table 3 (bottom) quantitatively illustrates data-type definition, reuse and composition in the three frontiers. The Direct Migration frontier contained 182 different data-types, and 73% of them were defined only once. In contrast, there were not duplicated data-types for the WSDL documents of the Indirect Migration frontier. Concretely, 104 data-types represented business objects, including 39 defined as simple XSD types (mostly enumerations) and 65 defined as complex XSD types. Finally, 131 extra data-types were definitions introduced to be compliant with the Web Service Interoperability standards (`http://www.ws-i.org/Profiles/BasicProfile-1.1.html`), which define rules for making Web Services interoperable among different platforms. Last but not least, 191 data-types were defined in the Assisted Migration frontier, including 118 definitions in the form of business objects (34 simple XSD types + 84 complex XSD types), and 73 definitions for WS-I compliance reasons.

With respect to data-type repetitions, the Direct Migration frontier included 182 data-types definitions of which 141 where unique. This means that 23% of the definitions were not necessary and could be substituted by other semantically compatible data-type definitions. The Indirect Migration frontier, on the other hand, included 235 *unique* data-types. The Assisted Migration frontier had 191 data-types, and 169 of them were unique. Then, our tool generated WSDL documents almost as good as the ones obtained after indirectly migrating the system completely by hand. Overall, the average data-type definitions per effective data-type across WSDL documents in the frontiers were 1.29 (Direct Migration), 1 (Indirect Migration), and 1.13 (Assisted Migration).

Moreover, the analyzed WSDL frontiers contained 182, 104, and 118 different definitions of business object data-types, respectively. The Indirect Migration frontier had fewer data-type definitions associated to business objects (104) than the Direct Migration frontier (182) and the Assisted Migration frontier (118), and therefore a better level of data model reutilization and a proper utilization of the XSD `complex` and `element` constructors for WS-I related data-types. However, the Assisted Migration frontier included almost the same number of business objects than the Indirect Migration frontier, which shows the effectiveness of the data-type derivation techniques of our tool.

Figure 2 illustrates how the WSDL documents of the Indirect Migration and Assisted Migration frontiers reused the data-types. The Direct Migration frontier was left out because its WSDL documents did not share data-types among them. Unlike the Assisted Migration graph, the Indirect Migration frontier has a reuse graph without *islands*. This is because using our tool is not as good as exhaustively detecting candidate reusable data-types by hand. Nevertheless, only 2 services were not connected to the bigger graph, thus our tool adequately exploits data-type reuse.

3.3 Anti-pattern analysis

We performed an anti-pattern analysis in the WSDL documents included in the three frontiers. We found the following anti-patterns in at least one of the WSDL documents:

– Inappropriate or lacking comments [11] (AP_1): A WSDL operation has no comments or the comments do not effectively describe its purpose.

This is a preprint of the article: "G. Salvatierra, M. Crasso, C. Mateos and A. Zunino: "Towards a Computer Assisted Approach for Migrating Legacy Systems to SOA". Lecture Notes in Computer Science (12th International Conference on Computational Science and Its Applications - ICCSA 2012). Vol. 7336, pp. 484-497. Springer-Verlag. 2012. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_36

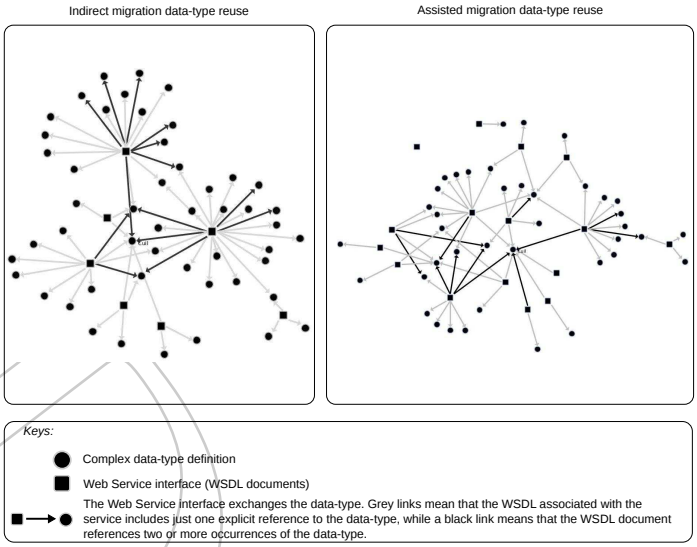

Fig. 2. Data-type reuse in the Indirect Migration and Assisted Migration frontiers.

- Ambiguous names [5] (AP_2): WSDL operation or message names do not accurately represent their intended semantics.
- Redundant port-types (AP_3): A port-type is repeated within a WSDL document, usually in the form of one port-type instance per binding type (e.g. HTTP or SOAP).
- Enclosed data model (AP_4): The data model in XSD describing input/output data-types are defined within a WSDL document instead of separate XSD files, which makes data-type reuse across several Web Services very difficult.
- Undercover fault information within standard messages [6] (AP_5): Error information is returned using output messages rather than built-in WSDL *fault* messages.
- Redundant data models (AP_6): A data-type is redundantly defined in a document.
- Low cohesive operations in the same port-type (AP_7): Occurs in services that place operations for checking service availability (e.g. "ping", "isAlive") of the service and operations related together with its main functionality into a single port-type.

Table 4 summarizes the outcomes of the analysis. When an anti-pattern affected only a portion of the WSDL documents in a frontier, we analyzed which is the difference between these WSDL documents and the rest of the WSDL documents in the same frontier. Hence, the inner cells present under which circumstances the former situation applies. Since spotting some of the anti-patterns (e.g. AP_1 and AP_2) is inherently subjective [7], we performed a peer-review methodology to prevent biases.

The Direct Migration frontier was affected by more anti-patterns than the Assisted Migration frontier, while the Indirect Migration frontier was free from anti-patterns. The first two rows describe anti-patterns that impact on services comments/names. These anti-patterns affected the Direct Migration frontier since all WSDL documents included in it were derived from code written in COBOL, which does not offer a standard way to indicate from which portions and scope of a code existing comments can be extracted

Table 4. Anti-patterns analysis: Obtained results.

Anti-pattern/Frontier	Direct Migration	Indirect Migration	Assisted Migration
AP_1	Always	**Never**	When the original COBOL programs use control couples
AP_2	Always	**Never**	When the original COBOL programs use control couples
AP_3	When supporting several protocols	**Never**	**Never**
AP_4	Always	**Never**	**Never**
AP_5	Always	**Never**	**Never**
AP_6	When two operations use the same data-type	**Never**	**Never**
AP_7	**Never**	**Never**	When several related programs link to non-related operations

and reused. Besides, COBOL names have length restrictions (e.g. up to 4 characters in some flavors). Therefore, names in the resulting WSDL documents were too short and difficult to be read. In contrast, these anti-patterns affected WSDL documents in Assisted Migration frontier only for those COBOL programs using control couples, because properly naming and commenting such couples is a complex task [12].

The third row analyzes the anti-pattern that ties abstract service interfaces (WSDL port-types) to concrete implementations (WSDL bindings), and as such hinders black-box reuse [8]. We observed that this anti-pattern was caused by the WSDL generation tools supporting the migration process that resulted in the Direct Migration frontier. Unless properly configured, these tools by default produce redundant port-types when deriving WSDLs from COBOL programs. Likewise, the fourth row describes an anti-pattern that is generated by this tool as well as many similar tools, which involves forcing data models to be included within the generated WSDL documents, making cross-WSDL data-type reuse difficult. Alternatively, neither the Indirect Migration nor the Assisted Migration frontiers were affected by these two anti-patterns.

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 frontier resulted from the original COBOL programs that used the same data output record for returning both output and error information. In contrast, the WSDL documents of the Indirect Migration frontier and the Assisted Migration frontier had a proper designed error handling mechanism based on standard WSDL *fault* messages.

The anti-pattern described in the sixth row is related to badly designed data models. Redundant data models usually arise from limitations or bad use of WSDL generation tools. Therefore, this anti-pattern only affected the Direct Migration frontier. Although there was not intra WSDL data-type repetition, the Assisted Migration frontier suffered from inter WSDL repeated data-types. For example, the error data-type –which consists of a fault code, a string (brief description), a source, and a description– was repeated in all the Assisted Migration WSDL documents because the data-type was derived several times from the different various programs. This problem did not affect the Indirect Mi-

gration frontier since its WSDL documents were manually derived by designers after having a big picture of the legacy system.

The last anti-pattern stands for having no semantically related operations within a WSDL port-type. This anti-pattern neither affected the Direct Migration nor the Indirect Migration frontier because in the former case each WSDL document included only one operation, whereas in the latter case WSDL documents were specifically designed to group related operations. However, our approach is based on an automatic heuristic that selects which operations should go to a port-type. In our case study, we found that when several related operations used the same unrelated routines, such as text-formatting routines, our assisted approach to migration suggested that these routines were also candidate operations for that service. This resulted in services that had port-types with several related operations but also some unrelated operations.

3.4 Required effort analysis

In terms of manpower, it took 1 day to train a developer on the method and tools used for building the Direct Migration frontier. Then, a trained developer migrated one transaction per hour. Thus, it only took 5 days for a trained developer to build the 32 WSDL documents. Instead, building the Indirect Migration frontier demanded one year plus one month, and 8 software analysts, and 3 specialists for migrating the same 32 transactions, from which 5 months were exclusively dedicated to build its WSDL documents.

We also have empirically assessed the time needed to execute our semi-automatic heuristics. The experiments have been run on a 2.8 GHz QuadCore Intel Core i7 720QM machine, 6 Gb RAM, running Windows 7 on a 64 bits architecture. To mitigate noise introduced by underlying software layers and hardware elements, each heuristic has been executed 20 times and the demanded time was measured per execution. Briefly, the average execution time of an heuristic was 9585.78 ms, being 55815.15 ms the biggest achieved response time, i.e. the “Detect semantically similar services and operations” was the most expensive heuristic in terms of response time.

Furthermore, we have assessed the time demanded for manually applying the OO refactorings proposed by the approach on the Direct Migration frontier. To do this, one software analyst with solid knowledge on the system under study was supplied with the list of OO refactorings produced by the approach. It took two full days to apply the proposed OO refactorings. In this sense, the approach suggested to “Expose 14 shared programs as services”, “Remove 7 redundant operations, “Improve the cohesion of 14 services”, and to “Improve error handling definition”, “Improve names and comments”, and “Improve business objects definition” from all the migrated operations. It is worth noting that OO refactorings have been applied at the interface level, i.e. underlying implementations have not been accommodated to interface changes. The reason to do this was we only want to generate a new SOA frontier and then compare it with the ones generated by the previous two migration attempts. Therefore, modifying interfaces implementation, which would require a huge development and testing effort, would not contribute to assessing service interfaces quality.

4 Related work

Migration of mainframe legacy systems to newer platforms has been receiving lots of attention as organizations have to shift to distributed and Web-enabled software solutions. Different approaches have been explored, ranging from wrapping existing systems with Web-enabled software layers, to 1-to-1 automatic conversion approaches for converting programs in COBOL to 4GL. Therefore, current literature presents many related experience reports. However, migrating legacy systems to SOA, while achieving high-quality service interfaces instead of just "webizing" the systems, is an incipient research topic.

Harry Sneed has been simultaneously researching on automatically converting COBOL programs to Web Services [13] and measuring service interfaces quality [14]. In [13] the author presents a tool for identifying COBOL programs that may be *servified*. The tool bases on gathering code metrics from the source to determine program complexity, and then suggests whether programs should be wrapped or re-engineered. As such, programs complexity drives the selection of the migration strategy. As reported in [13], Sneed plans to inspect resulting service interfaces using a metric suite of his own, which comprises 25 quantity, 4 size, 5 complexity and 5 quality metrics.

In [15] the authors present a framework and guidelines for migrating a legacy system to SOA, which aims at defining a SOA frontier having only the "optimal" services and with an appropriate level of granularity. The framework consists of three stages. The first stage is for modeling the legacy system main components and their interactions using UML. At the second stage, service operations and services processes are identified. The third stage is for aggregating identified service elements, according to a predefined taxonomy of service types (e.g. CRUD Services, Infrastructure services, Utility services, and Business services). During the second and third stages, software analysts are assisted via clustering techniques, which automatically group together similar service operations and services of the same type.

5 Conclusions and Future work

Organizations are often faced with the problem of legacy systems migration. The target paradigm for migration commonly used is SOA since it provides interoperability and reusability. However, migration to SOA is in general a daunting task.

We proposed a semi-automatic tool to help development teams in migrating COBOL legacy systems to SOA. Our tool comprises 10 heuristics that detect bad design and implementation practices in legacy systems, which in turn are related to some early code refactorings so that services in the final SOA frontier are as clear, legible and discoverable as possible. Through a real-world case study, we showed that our approach dramatically reduced the migration costs required by indirect migration achieving at the same time a close service quality. In addition, our approach produced a SOA frontier much better in terms of service quality than that of "fast and cheap" approach to migration (i.e. direct migration). The common ground for comparison and hence assessing costs and service quality was some classical software engineering metrics, data-type related metrics, and a catalog of WSDL anti-patterns [8] that hinder service reusability.

At present, we are refining the heuristics of our tool to improve their accuracy. Second, we are experimenting with an RM-COBOL system comprising 319 programs

and 201,828 lines of code. In this line, we will investigate whether is possible to adapt all the evidences heuristics to be used with other COBOL platforms. Lastly, even when direct migration has a negative incidence in service quality and WSDL anti-patterns, there is recent evidence showing that many anti-patterns are actually introduced by the WSDL generation tools used during migration [16]. Our goal is to determine to what extent anti-patterns are explained by the approach to migration itself, and how much of them depend on the WSDL tools used.

References

1. Juan Manuel Rodriguez, Marco Crasso, Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Bottom-up and top-down COBOL system migration to Web Services: An experience report. *IEEE Internet Computing*, 2011. To appear.
2. Martin Bichler and Kwei-Jay Lin. Service-Oriented Computing. *Computer*, 39(3):99–101, 2006.
3. John Erickson and Keng Siau. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of Database Management*, 19(3):42–54, 2008.
4. Shing-Han Li, Shi-Ming Huang, David C. Yen, and Cheng-Chun Chang. Migrating legacy information systems to Web Services architecture. *Journal of Database Management*, 18(4):1–25, 2007.
5. M. Brian Blake and Michael F. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12(5):62–69, 2008.
6. Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Jack, and Brad A. Myers. Usability challenges for enterprise service-oriented architecture APIs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 193–196, Sept. 2008.
7. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Automatically detecting opportunities for Web Service descriptions improvement. In *Software Services for e-World*, volume 341, pages 139–150. SADIO - IFIP, Springer Boston, 2010.
8. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.
9. Cristian Mateos, Marco Crasso, Alejandro Zunino, and Marcelo Campo. 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)*, pages 2509–2513. ACM Press, 2010.
10. Martin Fowler. Refactorings in Alphabetical Order, 1999.
11. Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Rec.*, 34(1):24–32, 2005.
12. Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
13. Harry Sneed. A pilot project for migrating COBOL code to Web Services. *International Journal on Software Tools for Technology Transfer*, 11:441–451, 2009. 10.1007/s10009-009-0128-z.
14. Harry Sneed. Measuring Web Service interfaces. In *12th IEEE International Symposium on Web Systems Evolution*, pages 111 –115, sept. 2010.

15. Saad Alahmari, Ed Zaluska, and David De Roure. A service identification framework for legacy system migration into SOA. In *Proceedings of the IEEE International Conference on Services Computing*, pages 614–617. IEEE Computer Society, 2010.
16. Cristian Mateos, Marco Crasso, Alejandro Zunino, and José Luis Ordiales Coscia. Detecting WSDL bad practices in code-first Web Services. *International Journal of Web and Grid Services*, 7(4):357–387, 2011.