Opening Web Applications for Third Party Development: a Service-Oriented Solution

Mohamed Lamine Kerdoudi, Chouki Tibermacine, Salah Sadou

Noname manuscript No.
(will be inserted by the editor)

Opening Web Applications for Third Party Development: a Service-Oriented Solution

Mohamed Lamine Kerdoudi · Chouki Tibermacine · Salah Sadou

the date of receipt and acceptance should be inserted later

Abstract Web applications are nowadays prevalent software systems in our every-day's life. A lot of these applications have been developed for end-users only. Thus, they are not designed by considering future extensions that would be developed by third parties. One possible and interesting solution for opening these applications for such kind of extension development is to create and deploy Web services starting from these applications. In this paper, we present a method and a tool for semi-automatically creating Web service implementations from applications having Web interfaces. The proposed method generates operations that are published in Web services for each functionality provided by a Web application. In addition, it generates new operations starting from Web interfaces. Our approach goes further in the creation of services by generating executable orchestrations, as BPEL processes, starting from navigations in the Web interfaces of these applications and by providing BPMN choreography specifications starting from the collaborations between the generated Web services. We implemented and experimented our solution in the migration of three real-world Web applications towards Web service-oriented systems.

Keywords Web Application, SOA, Web Service, Service Composition and Application Migration

Mohamed Lamine Kerdoudi
Computer Science Department, University of Biskra, Algeria
E-mail: lamine.kerdoudi@gmail.com

Chouki Tibermacine
LIRMM, CNRS and Montpellier University, France
E-mail: tibermacin@lirmm.fr

Salah Sadou
IRISA, University of South Brittany, France
E-mail: salah.sadou@irisa.fr

1 Introduction

Web applications are software systems that are widely used since the early nineties and the emergence of the World Wide Web. They have gained a lot of popularity comparatively to Desktop applications, because of their ease of use, *via* Web browsers, whereas Desktop applications need sometimes heavy installations. These applications provide to their users Web interfaces through which they can submit data to server-side scripts and through which they can receive the processing results.

The majority of Web applications have been designed and deployed exclusively for end-users that are humans. They have not been considered as a possible basis for remote extensions by third parties. For doing so, third party developers have no other choice than manually programming HTTP requests and then parsing the HTML code returned by these Web applications. This represents a cumbersome task for a developer especially that in most cases the parsed HTML code is too long and verbose. In addition, HTTP requests need frequently a detailed customization and HTTP responses need careful handling (dealing with errors and redirections).

In this paper, we propose a method for creating Web services by analyzing the source code and the configuration files of Web applications. Our method helps the Web application providers in opening their software product for the development of extensions, without having to write the necessary code for doing so from scratch. The proposed method generates operations that are published in Web services for each functionality provided by a Web application (methods and functions in the server-side source code of the application). In addition, it generates new operations starting from Web UI interfaces. This makes it possible to pro-

vide parameterizable services starting from pages designed for human interactions.

Besides the generation of these individual Web services, the proposed method creates composite Web services. It creates executable orchestrations of the generated Web services starting from the navigations between Web interfaces of the analyzed application. These orchestrations are generated as BPEL [27] processes. They implement a coarse-grained functionality provided by the Web application, comparatively with the individual Web services that implement fine-grained functionality. In addition, the proposed method creates choreographies of the generated Web services starting from the collaborations between them. These choreographies are specified in BPMN [32]. They provide models of a high level of abstraction, which help third party developers in understanding the architecture of the henceforth Web-service application.

We implemented our method on a collection of Java Frameworks. Component-based Web applications built with Java EE and its frameworks like JSF are the input of our implementation and a set of Web services are provided as output. The choice of such technologies is motivated by the fact that they offer a structured organization of the source code of Web applications. This made easy the parsing performed in our approach in order to generate Web services.

The potential beneficiaries of this work are: i) the organization which holds the rights on the Web application and whose developers would use the proposed method: migrating a Web application of this organization towards a service-oriented one enables the organization to modernize its "patrimony" and to shift to a new paradigm (of service orientation); ii) third party developers: they will be able to develop new business processes, potentially with financial benefits, starting from the generated Web services. The development of this "ecosystem" (composed of third party developers) around the generated services should necessarily bring a return on investment for the organization holding the rights on the original Web application.

The remaining of the paper is organized as follows. Section 2 introduces an example which better illustrates the problem tackled by our work and which serves as a running example for illustrating our proposals throughout the paper. Section 3 introduces a formal description of the context of our work, which is composed of Web applications and Web service-oriented systems. In Section 4 we give an overview of the proposed approach and we describe in detail how to create individual Web services starting from the components of the application. Section 5 introduces how the generated individual Web services are assembled to build composite services. In Section 6 we apply our method on three real-world Web applications, and we use some measures to show the effectiveness of the proposed solution. Before concluding and presenting some perspectives to this work, we describe the related work in Section 7.

2 Illustrative Example

Our example is an e-shopping Web application that offers to customers the opportunity to purchase electronic devices. The seller offers also delivery capabilities to ensure the transportation of the purchased items. The shopping process begins when the customer enters keywords for searching products via a Web interface (HTML form) provided by the application. A set of items which are relevant to these keywords are provided and distributed on HTML pages. The customer can choose one or a set of items among the returned ones. The selected items are saved in a virtual cart and the total price is then calculated and provided via a Web interface. Once the customer finishes the shopping (s)he is asked to sign in or to register for a new account. At the end, before proceeding to checkout the delivery schedule is prepared and provided via another Web interface.

2.1 Problem Statement

Let us suppose now that a third-party developer would like to implement an extension to this Web application. This extension concerns services for the purchased items (insurance against theft, breakage, fire, etc). This extension provides first to customers an interface for searching products that are for sale by the original application. The customer selects a set of desired products and chooses the quantity for each one. The extended version of the application includes all the steps from the original version. However, it gives the opportunity to choose an insurance service and then integrate its cost to the final amount.

So, there are some functionalities needed by this extension that are already provided by the original application (eg. searching, payment and delivery schedule). Therefore, for implementing this extension it would be interesting for the third-party developer if (s)he can use functionalities which are provided by the original application instead of implementing them from scratch.

In order to implement this solution, the third-party developer should have access to the functionality provided by the Web application differently than *via* its Web interfaces. Indeed, if (s)he uses only these Web interfaces, (s)he should send from her(his) programs the

necessary HTTP requests, with customized parameters, and should then parse the returned HTTP responses. This parsing involves an analysis of the responses in order to look for some specific parts which are of interest. In our example, the third-party developer should implement a program that sends an HTTP request to the server hosting the Web application, with for example the reference(s) of the product(s) (chosen by the customer). The parsing should identify the price of the purchased items, among other elements. As stated in the introduction, this task is time-consuming, cumbersome and error prone. In addition, the developer should know the exact type and structure of the HTTP requests and responses.

Moreover, unfortunately, there is no means to directly publish some services of the application for third party development. Even if stubs can be generated and provided for client applications, these stubs are generally language-dependent (only Java clients can use stubs generated for EJBs) and cannot be published, as they are, in libraries of services. Besides, the EJB 3.x specification introduced some annotations to enable developers to publish some methods in a bean as services. However, this is possible only for individual methods and we cannot introduce annotations to create composition of operations which we found in real-world business logic. In addition, we cannot use these annotations to expose Web interfaces as Web services. The same observations can be made on Eclipse tools (WTP project), which allow to generate Web services starting from individual methods in Java classes.

2.2 Potential Web Services

One of the best solutions for the previous problem is to enable the Web developer to create starting from the Web application a set of Web services suitable for remote extensions. In the presented example, the created Web services could be: a `Searching` Service for searching items, a `Cart` Service to manage a virtual shopping cart, an `Account` Service used to sign in or to register for a new account, a `Delivery` Service and a `Payment` Service.

The application extension scenario introduced previously can easily be implemented by remotely invoking the Web service operations. The extension of the shopping Web application can even be built simply as a BPEL process by invoking the `Searching` Web service using the reference of the chosen product as input. The returned price is added to the insurance's price.

The BPEL process makes the payment of the purchased products (without insurance) from the original Web application by invoking the generated `Payment` Web service. After that, to pay the insurance costs, the BPEL process invokes an operation that is implemented by the third party developer. At the end of this process, it invokes the `Delivery` Service.

3 Web applications and Service oriented Systems

Before presenting the details of our approach and how Web service-oriented systems can be derived from Web applications, we define in this section the different concepts used in this work. Indeed, we introduce formal descriptions of what composes Web applications and Web service oriented systems. These formal descriptions provide a better understanding of both kinds of software systems. In addition, this enables an accurate presentation of the processing performed on Web applications to generate Web service systems.

3.1 Web application Model

We adapt here the generic model of Briand *et al.* [**?**] to represent Web applications and their elements as a directed graph expressed using a set-theoretic notation. This graph is expressed as a pair (E, R), where E symbolizes a set of software entities found in a Web application, namely, `client side artifacts` and `server side artifacts`. R is a binary relation on $E(R \subseteq E \times E)$. It corresponds to the relationships between Web application elements, representing both structural and behavioral dependencies. Figure 1 gives an example of such a representation for an imaginary Web application.

Definition 1 *Representation of a Web application*

A Web application is represented as a pair (E, R), where

- *$E = CSA \cup SSA$ with:*
  - *CSA is the set of all client side artifacts which are HTML Web pages (WP in Figure 1) and Client Scripts (CS in Figure 1).*
  - *SSA is the set of all server side artifacts which are Server Scripts (SeS in Figure 1) and Server Classes (SC in Figure 1)*
- *R is the set of all common and possible relationships between artifacts of Web applications with:*
  - *$(CSA \times SeS)$ corresponding to relationships between static Web pages and server side scripts (such as `submit`, `build`, `redirect`, `clientScriptRequest`, among other relationships in Figure 1)*

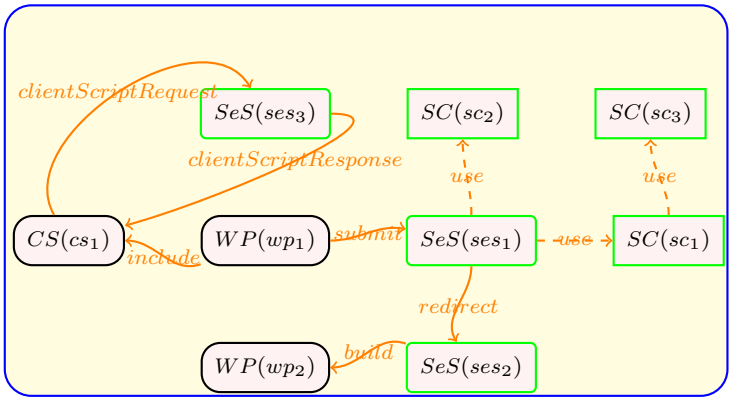

Fig. 1 Structure of a Web application

- *(SeS × SC) corresponding to relationships between server scripts and server classes (eg. (ses_1 **use** sc_1) in Figure 1)*
- *(SeS × SeS) corresponding to relationships between server scripts (eg.* **redirect** *in Figure 1)*
- *(SC × SC) corresponding to relationships between server classes (eg. (sc_1* **use** *sc_3) in Figure 1)*
- *(CSA × CSA) corresponding to relationships that exist between client side artifacts (eg. (wp_1* **include** *cs_1) in Figure 1)*

For instance, for the example given in Figure 1 we have:

- $CSA = \{ wp1, wp2, cs1\}$
- $SSA = \{ ses_1, ses_2, ses_3, sc_1, sc_2, sc_3\}$
- $R = \{(wp_1$ `include` $cs_1)$, $(ses_2$ `build` $wp_2)$,...$\}$

3.1.1 Representation of a Web Interface

A Web (user) interface (UI) may be formally defined as a sub-graph of the graph representing the Web application to which it belongs. Thus, a Web UI is defined by a pair (E_{wui}, R_{wui}) such as: $E_{wui} = (CSA_{wui} \cup SSA_{wui})$ with $(CSA_{wui} \subseteq CSA) \wedge (SSA_{wui} \subseteq SSA) \wedge (R_{wui} \subseteq R)$

A Web UI consists of server pages, client static pages and client built pages. The server pages are deployed on the Web server and could manipulate some server classes; client static pages have a static content which is composed of HTML tags; the content of client built pages is generated on the fly by the server pages after processing user's requests.

3.1.2 Example of a Web Interface Sub-Graph

Considering our example of the e-shopping application. In the Cart Web UI the total price of all saved items in the user's cart is calculated and presented to users via a Web interface.

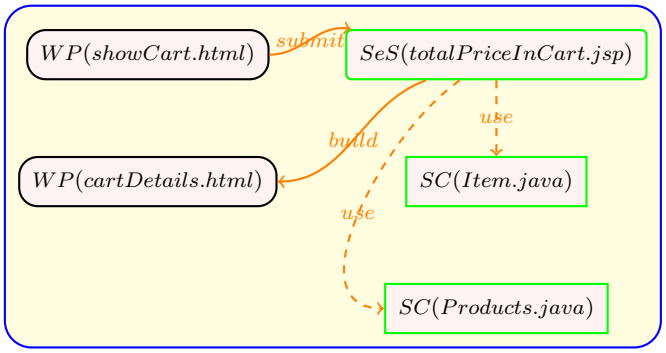

Fig. 2 A sub-graph representing the Cart Web Interface

Figure 2 shows a sub-graph that represents the Cart Web UI [1] where,

- $WP = \{ showCart.html, cartDetails.html\}$
- $SeS = \{totalPriceInCart.jsp\}$
- $SC = \{ Item.java, Products.java \}$
- R=$\{(totalPriceInCart.jsp$ `use` $Item.java)$,... $\}$

In this sub-graph, the user can access the cart through the client page *showCart.html*. In order to calculate the total price of the products in the cart, the user can submit data (such as product references and quantities) to the server script located in the JSP page *totalPriceInCart.jsp*. As a result, the total price is displayed to the user via the client page *cartDetails.html*.

3.1.3 Properties of server side artifacts

We define a set of structural and behavioral properties related to `server side artifacts` as follow:

- `Request Parameters`: are the data entered by users when manipulating a Web interface and which are processed by a server side script.

[1] This sub-graph is used as an illustrative example throughout this paper

For each element $ses \in SeS$,
$RP(ses)$ is the set of request parameters which are processed by the server script ses.
- `Environment Objects`: Each server side script could manipulate a set of environment objects such as session variables, cookies and business objects in order to store and share user's data.
  For each element $ses \in SeS$,
  $EO(ses)$ is the set of environment objects which are manipulated by the server script ses.
- `Produced Contents`: For each element $ses \in SeS$, $PC(ses)$ is the set of produced contents by a server side script as a result of processing user's requests.
- `Statements of Server Scripts`: For each $ses \in SeS$, $Stats(ses)$ is the set of all statements declared in ses.
- `Methods of Server Classes`: For each $sc \in SC$, $M(sc)$ is the set of all methods declared in a server class sc.
- `Method parameters`: Each server method has a set of parameters, where
  For each method $m \in M(sc)$,
  $IPar(m)$ is the set of input parameters of m and $OPar(m)$ is the set of output parameters of m.
- `Navigation Condition`: $NC(wp)$ represents the associated navigation condition to a Web page wp. It states that the user action navigates dynamically from the Web page wp to another page. In most web applications, navigation is not static. The page flow does not just depend on which button the user clicks, but also on the input value that (s)he introduces. For example, submitting a login page may have two outcomes: success or failure. The outcome depends on a computation (result of reference method invocation), namely, whether the username and password are valid.

The presented model and the set of definitions are used later throughout the paper.

3.2 Web Service Oriented System Model

Perepletchikov *et al.* [**?**] extended the generic model proposed by Briand *et al.* [**?**] and proposed a model covering structural and behavioral properties of the design artifacts in service-oriented systems. We adapt this model for representing the generated Web Service oriented application as a graph. In this graph, the WSDL files are used as service interfaces and Object-oriented (OO) classes are implementations for the primitive Web services. BPEL processes are used as implementations of the generated Web service orchestrations.

Fig. 3 shows an example of a graph representing the software entities of an imaginary Web service-oriented application.

Definition 2 *Representation of a Web Service-oriented System*

A Web service-oriented system is represented as a pair (E_{sos}, R_{sos})*, where*

- $E_{sos} = SI \cup BP \cup C$;
- SI *is a set of all service (WSDL) interfaces;*
- BP *is a set (possibly empty) of all BPEL processes that implement the WSDL service interfaces of the Web service orchestrations;*
- C *is a set of all OO classes that implement WSDL service interfaces of primitive Web services ;*
- R_{sos} *is the set of all common and possible relationships between the sets* SI*,* BP *and* C*. So,* $R_{sos} =$ `IIR` $\cup$ `ISR` $\cup$ `WSR` *with,*
  - `IIR` *(Interface Implementation Relationship) represents relationships between service interface and service implementation elements. A Service interface could be implemented using OO classes and/or business processes.*
  - `ISR` *(Internal Service Relationship) represents relationships between classes. Two classes in a given service could have a dependency relationship when an object of the first class invokes the methods (on objects) of the second class.*
  - `WSR` *(Web Service Request Relationship) represents relationships between a class (or a Business process) of a particular service and a service interface of another service. A class (or BPEL process) can invoke the operations defined in the service interface of another service.*

For instance, for the graph given in Figure 3 we have:

- $SI = \{si_1, si_2\}$
- $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$
- $BP = \{bpel_1\}$
- R= {(si_1 `IIR` c_1), (c_1 `ISR` c_2), (c_3 `WSR` si_2),...}

A Web service may be formally defined as a subgraph of the graph representing the Web service-oriented system to which it belongs. Thus, a Web service is defined by a pair (E_s, R_s): $E_s = (SI_s \cup BP_s \cup C_s)$ where $(SI_s \in SI) \wedge (BP_s \subseteq BP) \wedge (C_s \subseteq C) \wedge (R_s \subseteq R)$ with,

- A Web service has only a single service interface SI_s
- Each Web service exposes a set of operations, where
  - For each element $e \in SI \cup BP \cup C$,
    $O(e)$ is the set of operations of e.

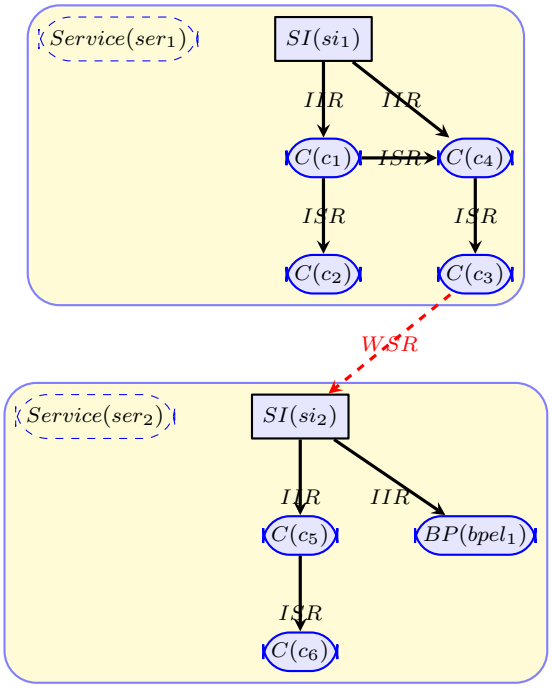

Fig. 3 Structure of a Web Service-oriented System

- For each operation $o \in O(e)$,
  $IParam(o)$ is the set of input parameters of o;
  $OParam(o)$ is the set of output parameters of o.
- For each operation $o \in O(e) \wedge e \in C$,
  $Code(o)$ is the set of all statements of o.

3.2.1 Example of Web service-oriented system graph

Returning to the Cart Web UI presented previously (Section 3.1.2), Fig. 4 shows a graph $SOS_{sos1} = (E_{sos1}, R_{sos1})$ that represents the Web services which could be generated starting from this Web UI.

- $SI = \{wsdlCart, wsdlProducts\}$
- $C = \{CartService, Item, ProductsService\}$
- $BP = \{\}$
- $R = \{(wsdlCart$ `IIR` $CartService)$, $(CartService$ `WSR` $wsdlProducts)$,...$\}$

In this sub-graph, $Service(ser1)$ represents the generated Web service starting from server scripts in the Cart Web UI and $Service(ser2)$ represents the generated Web service starting from the server class *Products.java*. The WSR relationship represents an invocation to an operation published in $wsdlProducts$ interface from the source code of the class $C(CartService)$. This relationship represents an invocation from a server script in the Cart Web UI to a method (this method is exposed later as a Web service operation) located in the *Products.java* class.

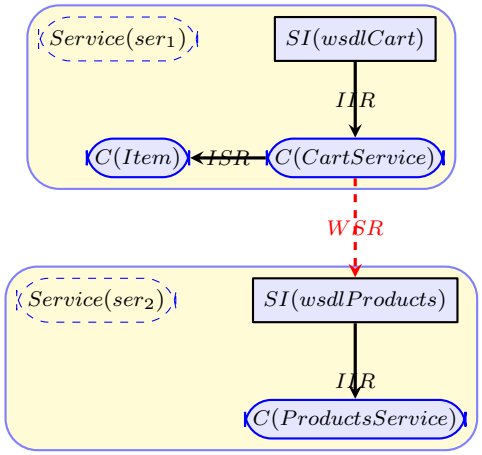

Fig. 4 A sub-graph that represents the generated Web services from the Cart Web Interface (SOS_1)

4 Proposed Approach

This section covers the proposed solution for migrating Web applications toward Web service-oriented systems. First, we give an overview of the proposed approach. Then, we present in detail each step in the process.

4.1 General Overview

The creation of Web services from Web applications is a semi-automatic multi-step process. This is illustrated in Fig. 5. The dashed boxes in the process represent steps where the developer is involved. This process begins by receiving from the developer as input the source code and the configuration files of the Web application to be analyzed. A set of primitive and composite Web services are provided as output.

To present the details of our approach accurately, we use the previous formal definitions to represent the input Web application as a pair (E, R) and the desired Web service oriented solution as another pair (E_{sos}, R_{sos}). Moreover, we show how to generate the Web service oriented application as a mapping from a Web application graph to a Web service-oriented system graph.

The algorithm 1 introduces a sequence of the main functions and procedures used to apply this mapping. In this algorithm we follow the same logic of steps in Fig. 5. First, each element in a Web application is statically parsed to identify the potential set of operations with their input and output messages. The operations can be identified starting from existing methods in server classes (see Line 3) and from the server

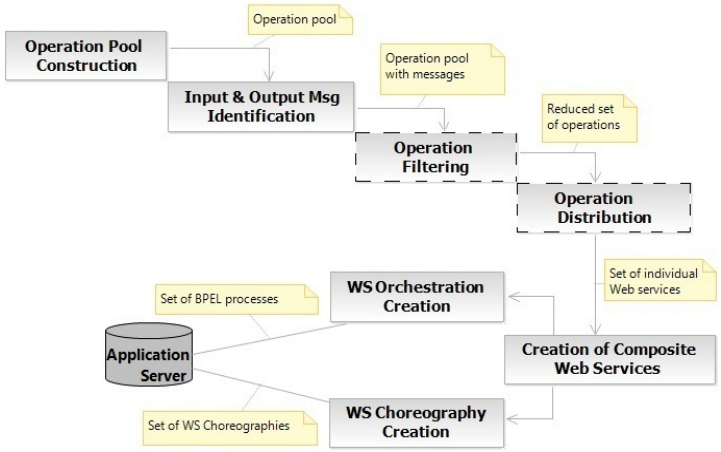

Fig. 5 The Proposed Process

scripts that provide functionalities via Web UIs (see Line 5). Then, all operations that must not be published in Web services are eliminated. This needs the contribution of the developer. (S)he should manually remove these unwanted operations. Besides, we provide to the developer a way for specifying constraints so that the operations that are recurrently unwanted can be automatically removed (see Line 9). After that, the remaining operations are grouped in Web services (see Line 10) and represented as sub-graphs of the output graph. The dependencies between Web services are then identified and represented using the WSR relationships (see Line 11). At the end, a set of Web service orchestrations are created (see Line 12) and added as additional sub-graphs of the output graph. Hereinafter we detail each function in the process.

Algorithm 1 From a Web application model to a Web service-oriented system model

Input: Web application model $WA = (E, R)$
Output: Web service-oriented system model $SOS = (E_{sos}, R_{sos})$
1: **for all** $e \in E$ **do**
2: **if** $e \in SC$ **then**
3: $s = identifyExistingOperations$(e)
4: **else if** $e \in SeS$ **then**
5: $s = identifyOperationsFromWebInterfaces$(e)
6: **end if**
7: add (s, SOS)
8: **end for**
9: $filterUnwantedOperations(SOS, oclConstraints)$
10: $distributeOperations(SOS)$
11: $createChoreographies(SOS)$
12: $createBPELProcesses(SOS)$
13: **return** SOS

4.2 Operation Pool Construction

A pool of operations is constructed by parsing the Web application's contents. Two kinds of operations are created.

4.2.1 Identification of Existing Operations

These operations are generated starting from the existing methods and functions in the back-end components of the Web application. To do so, we statically parse different elements (e.g. classes or server scripts) of the Web components forming the transformed application. All methods in classes and functions in scripts are prepared to be considered as potential operations in Web services. For instance, in the example presented in section 3.1.2 the public methods that are declared in the *Products.java* class are transformed into new operations. Algorithm 2 shows how to create a service containing operations that are generated starting from existing public methods in a server class.

Algorithm 2 Identify Existing Operations

1: **function** IDENTIFYEXISTINGOPERATIONS($serverClass : SC$)
2: $s =$ `create` $Service$: s=(SI_s, BP_s, C_s, R_s) ▷ s is subgraph of a service
3: $wsdl =$ `create` $ServiceInterface : wsdl \in SI_s$
4: $c =$ `create` $Class : c \in C_s$
5: $r =$ `create IIR` $: r = (wsdl$ `IIR` $c) \wedge r \in R_s$
6: **for all** $m \in$ `M`$(serverClass)$ **do**
7: **if** $isPublic(m)$ **then**
8: $createNewOperation(s, c, m, wsdl)$
9: **end if**
10: **end for**
11: **return** s
12: **end function**

Therefore, the service interface and its implementation class are created here (see Lines 3 to 5 in Algorithm 2).

Algorithm 3 gives the details of how to create operations. Each access to global variables or attributes from the source code of the identified methods and functions is transformed into additional parameters (see Line 5 in Algorithm 3). This makes these operations stateless. Moreover, each server class and all their dependent internal classes (which does not publish operations) are grouped together to form a Web service (see Lines 7 to 12 in Algorithm 3).

Algorithm 3 Create New Operation

1: **function** createNewOperation($s : SOS, c : C_s, m : M(c), wsdl : SI_s$)
2: $op =$ `create` $Operation : op \in O(c) \wedge op \in O(wsdl)$
3: $IParam(op) = IPar(m)$
4: $OParam(op) = OPar(m)$
5: $IParam(op) = IParam(op) \cup UsedGlobalVarIn(m)$
6: $Code(op) = Stats(m)$
7: **for all** $usedClass \in getUsedClassesIn(Code(op))$ **do**
8: **if** $usedClass$ does not publish operations **then**
9: $c_1 =$ `create` $Class : c \in C_s$
10: $isr =$ `create ISR` $: isr = (c$ `ISR` $c_1) \wedge isr \in R_s$
11: **end if**
12: **end for**
13: **end function**

4.2.2 Creation of New Operations from Web Interfaces

The main entities of a Web application are the Web user interfaces that consist of server's pages and client's pages. Through client pages, the end-users can submit data to server-side scripts and through which they can receive the processing results.

The public functionalities which are accessible via Web interfaces from end-users are transformed into new operations. These operations have as parameters the data entered by users when manipulating the Web interface (data entered in forms, for example), and have as output the results returned by the scripts processing the data entered by the users. In other words, the code present in programs executed at the server-side (JSP or PHP scripts, for example) is grouped within new operations and formatted to be executed as stand-alone code. For example, all the code present in scriptlets of a JSP page which implements a provided functionality to users is grouped and formatted within a single operation. For instance, in the Web interface presented in section 3.1.2, the code present in scriptlets of $totalPriceInCart.jsp$ server page is formatted within a new operation exposed by the $wsdlCart$ interface in Fig. 4. More details about this transformation is given in the next section.

Therefore, the Web interfaces that use a secure protocol such as TSL(SSL) [**?**] to protect the exchanged messages content or to authenticate the client by using a clients public key certificate are migrated toward a secure Web service that uses the same protocol. Indeed, the source code that implement the TSL(SSL) protocol in the Web interface is formatted to be a secure Web service. In this way, our approach will not weaken the migrated Web application security level.

Besides, a Web interface may include a Frameset composed of one or more frames, and in each frame, there is a content which could be dynamically loaded from elsewhere (Web intefaces, texts, images, etc.). For example, the `<iframe src="URL">` tags are used to embed another content (HTML, JSP, PHP...) within a given HTML, JSP or PHP document. For such tags, the inline frame (document) sometimes corresponds to a Web interface, which has been transformed into a Web service. In this case, an invocation to this Web service is added in the source code of the operation created starting from the currently analyzed Web interface. This invocation allows to retrieve some data from the server.

Algorithm 4 shows how to create a Web service subgraph starting from a Web interface sub-graph. First, the service interface and its implementation class are created (see Lines 3 to 5). After that, a new operation is created from the Web interface (see Lines 6 to 9). This operation and its internal dependent classes are grouped within a Web service (see Lines 10 to 15). In other words, for each dependent class we create a node of type C_s. This node is connected to the class that implements the service with an ISR relationship. For example, in Fig. 4 the $C(Item.java)$ class is an internal class used by the $C(CartService)$ class. Additional operations are also generated from the existing methods in the server page (see Lines 16 to 20).

4.3 Input and Output Message Generation

Based on the result of the first step, the input and output messages related to each operation in Web services are identified and generated starting from the parsed elements in the Web application: i) For operations in classes and other structured code elements, the parameters and the returned values are formatted as (respectively, input and output) SOAP messages (see Lines 3 and 4 in Algorithm 3); ii) The saved data in HTTP requests and HTTP responses are parsed to extract new input and output messages (see Lines 7 and 8 in the

Algorithm 4); iii) The used environment objects (session variables, cookies and business objects) by the Web interface are considered as input and output messages (see Lines 7 and 8 in Algorithm 4).

Algorithm 4 Identify Operations From Web Interfaces

```
1: function IDENTIFYOPERATIONSFROMWEBINTERFACES(serverClass : SeS)
2:     s = create Service : s=(SI_s, BP_s, C_s, R_s)        ▷ s is subgraph of a service
3:     wsdl = create ServiceInterface : wsdl ∈ SI_s
4:     c = create Class : c ∈ C_s
5:     r = create IIR : r = (wsdl IIR c) ∧ r ∈ R_s
6:     op = create Operation : op ∈ O(c) ∧ op ∈ O(wsdl)
7:     IParam(op) = RP(ses) ∪ EO(ses)
8:     OParam(op) = PC(ses) ∪ EO(ses)
9:     Code(op) = Filter(Stats(m))
10:    for all usedClass ∈ getUsedClassesIn(Code(op)) do
11:        if usedClass does not publish operations then
12:            c_1 = create Class : c ∈ C_s
13:            isr = create ISR : isr = (c ISR c_1) ∧ isr ∈ R_s
14:        end if
15:    end for
16:    for all meth ∈ declaredMethodIn(ses) do
17:        if isPublic(m) then
18:            createNewOperation(s, c, meth, wsdl)
19:        end if
20:    end for
21:    return s
22: end function
```

4.3.1 Dealing with HTTP requests and HTTP responses

The code present in the server programs (e.g. server pages like JSP or PHP pages) is parsed to extract the input values received in the HTTP requests (by identifying the statements getting values from HTTP requests). Their types are deduced from the parsed code by analyzing type casts and other conversion statements. This is directly possible in statically typed scripting languages like JSP and C♯. For dynamically typed ones (like PHP or Python), we use external tools for type inference. In addition, the shared objects (such as `JavaBeans` instances), which are used across multiple Web interfaces, are considered as additional input parameters for the generated operation. In this way the saved data in these objects can be passed from an operation to another in order to compose them.

Furthermore, the contents produced by the server programs, which are viewed at the client side (this content is produced using statements such as: JSP expressions or `out.println(...)` for JSP and PHP's `echo()` or `print()` function calls), are considered as output values. The types of these values are extracted from the code and defined in the generated SOAP messages. The arguments of the `out.println(...)` or `echo(...)` methods can be variables, method invocations or expressions. For variables, we get their type from the parsed code. In the case of method invocations, the type of the generated output message corresponds to the returned type of the invoked method. The expressions can be a concatenation of texts and values of variables and/or method invocations. In this case, the values from method invocations and the variable accesses are extracted to be added as output values of the generated operation. The text is added as another output.

Let us consider the Cart Web UI of our example presented in Section 3.1.2. This interface allows users to calculate the total price of her/his items. Listing 1 shows an excerpt of the code present in the *totalPriceInCart.jsp* server script. Two input messages are identified starting from the `request.getParameterValues(...)` statements (see Lines 8 and 9). They correspond to the `references` of the selected items to be purchased and the `quantities` wanted by the user for each selected item. Another input message is extracted from the used `JavaBean` object `prods`. For the sub-graph created for this Web interface (see Fig. 2), we represent these values as: RP=$\{references, quantities\}$ and EO= $\{prods\}$.

```
1  <jsp:useBean class="shop.prod.Products" id="prods" scope
       ="page"/>
2  <%!
3      String[] references;
4      String[] quantities;
5      double totalPrice = 0;
6  %>
7  <%
8    references = request.getParameterValues("references");

9    quantities = request.getParameterValues("quantities");

10    if(references != null){
11      for(int i = 0; i < references.length; i++){
12    Item item = prods.getItemByReference(references[i]);
13       double unitPrice = item.getUnitPrice();
14       int quantity = Integer.parseInt(quantities[i]);
15       totalPrice = totalPrice + calculateTPrice(
              unitPrice, quantity);
16     }
17    }
18 %>
19 <%= totalPrice %>
```

Listing 1 An excerpt of the code present in the `Cart` Web interface

The type of the `references` input is an array of Strings. From the conversion statement (see Line 14) we have deduced that the type of `quantities` is an array of Integers.

Finally, we have deleted from the source code of the generated operation: the conversion, the cast and the `request.getParameterValues(...)` statements (Lines 8, 9 and 14 in Listing 1). The returned value of the generated operation is the result saved in the variable `totalPrice` and bound to the Web interface using a JSP expression (Line 19). This is used to create an output SOAP message of type `Double` (in our sub-graph, we have $PC = \{totalPrice\}$).

Applying the two first steps and making the necessary modifications for the previous source code, a new operation is created (see Listing 2). For that, four input parameters and a returned value are identified.

```
1  public double serviceTotalPriceInCart(String[]
       references, int[] quantities, shop.prod.Products
       prods){
2      double totalPrice = 0;
3      if(references != null){
4       for(int i = 0; i < references.length; i++){
5         Item item = prods.getItemByReference(references[i
              ]);
6         double unitPrice = item.getUnitPrice();
7         int quantity = quantities[i];
8       totalPrice = totalPrice + calculateTPrice(
              unitPrice, quantity);
9       }
10     }
11     return totalPrice;
12 }
```

Listing 2 An excerpt of the generated operation from the `Cart` Web interface

Fig. 4 shows the subgraph that represents the generated Web service from the `Cart` Web interface and the server class `Products`, where, the following values represent these services.

- $O(CartService)$ =$\{totalPriceInCart$, $calculateTPrice$,...$\}$
- $IParam(totalPriceInCart)$= $\{references$, $quantities$, $prods$ $\}$
- $OParam(totalPriceInCart)$ = $\{$ $totalPrice\}$
- $IParam(calculateTPrice)$=$\{unitPrice$, $quantity\}$
- $OParam(calculateTPrice)$ = $\{TPrice\}$
- $O(Products)$ = $\{getItemByReference$, $getItemDetails$,...$\}$

4.3.2 Handling Session Objects

Most of Web applications manage session variables in order to store and share the user's data when (s)he navigates from a Web interface to another one. To avoid losing this data and make possible using them in the composition of the generated Web services from the Web interfaces, we consider these values as additional input and output messages.

The user's data that is stored in these session objects could be used as constraints for accessing other Web interfaces when the user navigates in the application. Therefore, the output messages generated from the first Web interface are considered as the input to the generated services from the navigated Web interfaces. This ensures that these services are not freely accessible (preserve security). In order to generate these messages, we parse the code present in operations that use session variables.

In our illustrative example, the `addItem` Web interface is used to add a new item into a virtual cart. Listing 3 shows an example of using session variables to store information about the cart. The `quantity` and the `reference` are identified and considered as input messages. They are identified after the parsing of the `request.getParameter(...)` statements (as explained in the previous section). Now, statements such as `session.getAttribute(...)` (see Lines 2 and 11) return the objects bound to the name `'currentCart'` specified in this session. So, this object is transformed into an additional input. The session objects can be updated in the source code (see Line 18). For this reason, they are also considered as output messages of the generated operation. After analyzing the cast statements in Lines 2 and 11 we have deduced that the concrete type of the `'currentCart'` message is `Cart`.

```
1  <%
2    Cart currentCart =(Cart) session.getAttribute("
         currentCart");
3   // return the object bound with the name 'currentCart'
         in this session, or null if no object is bound
        under this name
4    Cart newCart = null;
5    if (currentCart == null){
6    //binds a new object 'newCart' to this session using
         the name "currentCart"
7         newCart = new Cart();
8         session.setAttribute("currentCart",newCart);
9    }
10   else {
11        newCart = (Cart) session.getAttribute("
              currentCart");
12   }
13  String StrQuantity = request.getParameter("quantity");
14  int quantity = Integer.parseInt(StrQuantity);
15  if (quantity > 0){
16        String reference = request.getParameter("
              reference");
17      // update the Cart and the object bound the this
            session
18        newCart.addItem(reference,quantity);
19   }
20   out.println(newCart.getTotalPrice())
21 %>
```

Listing 3 An excerpt of a server script present in the `addItem` Web interface

In addition, the parsing of statements that are used to bind an object to a session, such as: `session.setAttribute(...,...)`, generates additional output messages. For example, from the Line 8 we create an output message named `'currentCart'`. At the end, an additional output message corresponding to the calculated price is generated from the parsing of the statement `out.println(...)` (see Line 20). The type of this output corresponds to the returned type of `getTotalPrice` method that is `Double`.

Listing 4 shows an excerpt of the newly created operation. It allows to add a new item in the virtual cart. This operation receives three input messages (`reference`, `quantity` and `currentCart`) and returns a composed message that contains the new total price and the updated virtual cart. However, the statements that use session variables are removed from the code of this operation.

```
1  public AddItemOutput serviceAddItem(String reference,
       int quantity , Cart currentCart ){
2    Cart newCart = null;
3    if (currentCart == null){
4        newCart = new Cart();
5        currentCart = newCart;
6    }
7    else {
8       newCart = currentCart;
9    }
10   if (quantity >0){
11       newCart.addItem(reference,quantity);
12   }
13   return (new AddItemOutput(newCart.getTotalPrice(),
         currentCart));
14 }
```

Listing 4 An excerpt of the generated operation from `addItem` Web interface

4.3.3 Dealing with Cookies

Actually, the server can maintain information about user sessions in many ways such as using cookies. In our approach, the used set of cookies is considered as input and output messages. The statements which are used to access and to modify the saved cookies (e.g. in a JSP page, `request.getCookies()` and `response.addCookie(...)`) are identified and replaced in the body of the new generated operation with equivalent statements accessing these new messages. Listing 5 shows an example of using cookies to save information about user authentication in the `signIn` Web interface.

The parsing of this code has identified one input message which is defined starting from the `request.getCookies()` statement (see Line 2). This input value corresponds to a set of cookies that are used in the generated operation's code. This set of cookies is returned as output message of this operation in order to be used as input of another operation generated from another Web interface that uses these cookies. In this way the composition of these two operations would be easier.

```
1  <%
2    Cookie[] cookies = request.getCookies();
3    String email = "", password ="";
4    if( cookies != null ){
5         for(Cookie cookie : cookies){
6            if (cookie.getName().equals("email")){
7              email = cookie.getValue();
8            }
9            if(cookie.getName().equals("password")){
10             password = cookie.getValue();
11           }
12         }
13           AccountManager userId= new AccountManager();
14           if(userId.signIn(email,password)){
15                // ....
16           }
17     }
18  %>
```

Listing 5 An excerpt of code showing the using of cookies in the `signIn` Web interface

The generated operation has a body with the same code as the script shown above, except the statement at Line 2. This Line is replaced with a statement which is used for extracting the cookies from an object of type Collection (`obj.getCookies()`) received as an argument. In addition, a "`return`" statement is added at the end of the operation's body, which returns this object (`retun obj;`).

4.4 Operation Filtering

In this step, the identified pool of operations is filtered by eliminating the operations which are not suitable to be published in Web services. For example, the modern Web applications use Public and Private APIs. Thus, by this filtering task, we allow developers to eliminate all operations that are identified from Private APIs. The filtering cannot be fully automated and it needs the developer involvement. The developer is asked to choose among the selected operations those that are not interesting for a publication. A set of filtering expressions are made available to be used and enriched by the developer. Some kinds of operations are recurrent in most of applications. Therefore, the specified expressions could be reused by another developer in order to filter operations which are generated starting from other Web applications. The developer will not have to

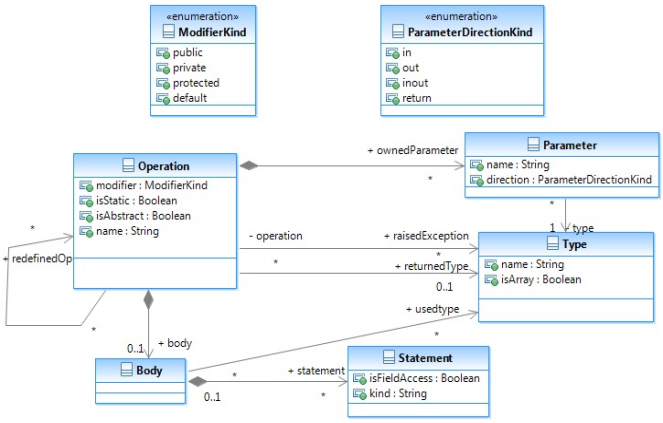

Fig. 6 The Operation Meta-model

specify them from scratch. These expressions are constraints that are checked on an Ecore [13] instance of a meta-model representing operations. These instances of the meta-model are automatically built by analyzing the operations' code. Constraints are Boolean expressions which are specified using OCL (Object Constraint Language [30]). OCL has been chosen because of its simplicity [6] and the existence of a good tool support (OCL Toolkit [10], Eclipse MDT/OCL [14], ...). The specified constraints navigate in the meta-model, which is illustrated in Figure 6. This meta-model is an excerpt of the UML meta-model (related to operations) [33] extended with some basic constructs.

The main meta-class in Figure 6 is `Operation`, which represents an identified operation from the code. The `Operation` meta-class is associated to a `Type` meta-class which represents the returned type of the operation. In addition, this operation could have a body and a set of parameters.

All constraints have as a context an instance of the `Operation` meta-class. An example of a constraint is given below:

```
context Operation inv :
not ((self.returnedType.name= 'void') and
(self.name.substring(1,3) = 'set') and
(self.ownedParameter->size() = 1) and
self.body.statement->exists( kind =
 'AssignmentStatement' and
 isFieldAccess = 'true')
```

In this example, all operations that represent field accessors (for example, setter methods) are eliminated.

4.5 Operation Distribution in Services

The extracted operations are distributed on multiple Web services based on the following criteria:

4.5.1 Grouping Criterion

We group operations based on the cohesion and coupling criteria. We argue that an optimal granularity is the key to a well-designed service. Service granularity generally refers to the performance and size of a service [**?**]. The data granularity is one of service granularity types [**?**]. It reflects the amount of data that is exchanged with a service. A good grouping of the identified operations in Web services has a positive impact on the data granularity. In our approach, the highly coupled and cohesive operations are grouped together in a single Web service. And, the low coupled operations are distributed on multiple services. This strategy of grouping reflects a low amount of data which is exchanged and reduces the communication overhead. Hence from the service quality viewpoint, we increase the performance and the maintainability of the generated services.

The cohesion of a service is assessed based the degree of the strength of functional relatedness of operations within a service. We measure the cohesion of a service by analyzing the static invocations between operations within that service. Several cohesion metrics have been proposed in the literature in order to measure the cohesion of a class in an object-oriented system [7]. We believe that one of these metrics can be used to evaluate the cohesiveness of the generated Web services. The $LCOM$ (Lack of Cohesion in Methods), TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) are one of the most used metrics to measure cohesion between public methods in a class. The problem with $LCOM$ metric is that such metric only helps in identifying the absence of cohesion rather than its presence [**?**]. On the contrary, we need in our work to measure the presence of cohesion in Web services. For this reason we use the TCC and LCC metrics to measure service cohesion. To do so, we start with a flat organization of operations (all operations are distributed in one Web service). Then, we calculate TCC and LCC to check the cohesiveness of this Web service. If the service is not cohesive, we split it into set of low coupling services and we check again the cohesiveness of each one of them. We repeat the measurement until the produced services are "quite cohesive".

To measure the TCC, we consider a Web service with N operations. NP is the maximum number of operation's pairs:$NP = [N * (N - 1)]/2$. The NDC is the number of direct connections between operations. Then TCC is defined as the relative number of directly connected operations: $TCC = NDC/NP$.

To measure LCC, we consider NIC is the number of indirect connections between operations (when two

operations are connected via other operations). LCC is defined as the relative number of directly or indirectly connected operations: $LCC = NDC + NIC/NP$.

According to [3], a class (a service in our case) is considered non-cohesive when $TCC < 0.5$ and $LCC < 0.5$. If $LCC = 0.8$ the class is considered "strongly cohesive". If $TCC = LCC = 1$ then the class is maximally cohesive, which means all methods are connected. In our approach, we experimentally tested these metrics and we found out that with $TCC \geqslant 0.5$ or $LCC \geqslant 0.5$ the obtained Web services are "quite cohesive".

Algorithm 5 shows how the grouping is performed. First, each group of operations is represented with a graph which is expressed as a pair (OP, CON), where OP symbolizes a set of operations. CON is a binary relation on $CON(\subseteq OP \times OP)$. It corresponds to the direct and indirect connections between operations.

The input of this algorithm is a group that contains all the identified operations. The output is a set of cohesive Web services.

Algorithm 5 Grouping Operations

```
1: function GOUPINGOPERATIONS(CL = (OP, CON)))
2:     TCC = calculateTCC(CL)
3:     if TCC ⩾ 0.5 then
4:         return TRUE
5:     else
6:         LCC = calculateLCC(CL)
7:         if LCC ⩾ 0.5 then
8:             return TRUE
9:         else
10:            if existExplicitGroups(CL) then
11:                explicitGroups = split(CL)
12:                for all group ∈ explicitGroups do
13:                    goupingOperations(group)
14:                end for
15:            else
16:                implicitGroups = getImplicitGroups(CL)
17:                for all group ∈ implicitGroups do
18:                    goupingOperations(group)
19:                end for
20:            end if
21:            return FALSE
22:        end if
23:    end if
24: end function
```

First, we try to find the best grouping (best level of cohesiveness) by measuring the TCC. If $TCC \geqslant 0.5$ we conclude that the Web service is "quite cohesive". In this case we do not need to calculate the LCC, because, the existing number of direct connections is enough, in order to know if service is cohesive or not. Now, in the case of $TCC < 0.5$, the indirect connections between operations that belongs to a service are used to assess the cohesiveness of that service. Hence, we need to calculate the LCC. If $LCC \geqslant 0.5$ we consider the Web service is "cohesive" although the $TCC < 0.5$. Now, if the $LCC < 0.5$ and $TCC < 0.5$, then the service is not cohesive. In this case, we split the service into a set of explicit groups of operations (where, there are no connections between these groups) (see Line 11). For each explicit group we invoke again the grouping algorithm. Now, if there is no explicit groups, we identify the implicit groups where there is a lowest coupling between them (see Line 16). And we repeat the measurement for each group.

For instance, the following operations are created starting from the e-shopping application:

- (op_1) : boolean serviceLogin(String userName, String paswword)
- (op_2) : boolean authenticate(String userName, String paswword)
- (op_3) : User getUserDetails(String userName)
- (op_4) : boolean userRegistration(String firstname, String lastname, String address, int mobile, String email, String password)
- (op_5) : Item[] serviceProducts()
- (op_6) : Item[] getAllItems()
- (op_7) : Item[] getItemsByUser(String userName)
- (op_8) : Item getItemDetails(String reference)
- (op_9) : Item getItemByReference(String reference)
- (op_{10}) : Double serviceTotalPriceCart(String[] references, int[] quantities)
- (op_{11}) : Double calculateTPrice(Double unitPrice, int quantity)

Fig. 7 shows the dependencies between operations where, dashed lines represent the indirect connections between a pair of operations and solid lines represent direct connections between them. The measurements give the following values:

- TCC(G1) = $\frac{6}{55} = 0.10$
- LCC(G1) = $\frac{6+6}{55} = 0.21$
- TCC(G2) = $\frac{3}{6} = 0.5$
- LCC(G2) = $\frac{3+2}{6} = 0.83$
- TCC(G3) = $\frac{3}{21} = 0.14$
- LCC(G3) = $\frac{3+4}{21} = 0.33$
- TCC(G4) = $\frac{1}{10} = 0.10$
- LCC(G4) = $\frac{1+4}{10} = 0.5$
- TCC(G5) = $\frac{1}{1}$ =1
- LCC(G5) = $\frac{1}{1} = 1$

We have started with the group G1 which is not cohesive. G1 is divided into two explicit groups (G2) and (G3). After that, (G4) and (G5) are created starting from (G3). Finally, the obtained cohesive Web services are: G2 (op1, op2, op3, op4), G4(op5, op6, op7, op8, op9) and G5(op10, op11).

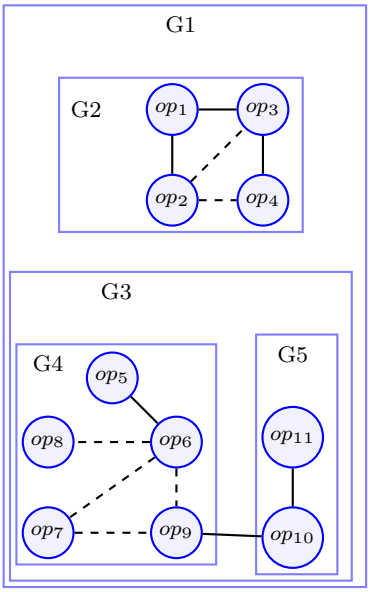

Fig. 7 Example of operations grouping

4.5.2 Spreading Criterion

Similar operations are spread out in different Web services. In this way, for users of an operation within a service, another service containing a similar operation can be easily and quickly found at the same provider (reliability).

In other words, Web services are exposed to errors and failures for many reasons, such as, the network is unreachable, the application server is unavailable or the service is not working properly. Hence, the reliability of the programs (it could be an orchestration of Web services) that invokes these services will be decreased. Several error-handling approaches are proposed in the literature such as [**?**], [**?**], [**?**] and [**?**]. Many of these approaches are based on finding a relevant service substitute that replaces the failed service. For example in previous works [2] and [**?**] the identification of the substitute is based on the measurement of similarity between service interfaces.

In this work, a solution based on a comparison of operation signatures has been used. The WSSim tool [40] allows to measure the similarity between operations by comparing the operations' names and input and output messages. Table 1 shows the similarity measurement results that are produced by WSSim for the operations that are depicted in Fig. 7. In this table we give a score of similarity (between 0 and 1) for all pairs of operations. The operations that have a similarity score that ranges between 0.80 and 1 are considered highly similar. According to the obtained similarity assessment, we consider that operations op_8 and op_9 are highly similar.

Table 1 Obtained similarity scores

	op_1	op_2	op_3	op_4	op_5	op_6	op_7	op_8	op_9	op_{10}	op_{11}
op_1	1	0.74	0.53	0.73	0.57	0.41	0.51	0.50	0.49	0.63	0.55
op_2		1	0.52	0.5	0.45	0.42	0.67	0.60	0.55	0.52	0.61
op_3			1	0.53	0.55	0.61	0.70	0.70	0.56	0.38	0.50
op_4				1	0.46	0.35	0.53	0.56	0.53	0.59	0.53
op_5					1	0.42	0.49	0.50	0.49	0.52	0.47
op_6						1	0.63	0.64	0.60	0.34	0.49
op_7							1	0.76	0.79	0.38	0.51
op_8								1	0.86	0.48	0.56
op_9									1	0.59	0.53
op_{10}										1	0.65
op_{11}											1

After the calculation of cohesion and similarity between the different operations, the next step is to distribute these operations on Web services. In the current implementation, we assist the developer for giving a new organization based on the obtained results from the cohesion and similarity values. The proposed organization could then be manually updated by the developer. For example, we decide to move the operation (op_8) from the Web service G4 into G5. This moving does not have a negative impact on the cohesiveness of the obtained services.

5 Generation of composite Web Services

In this step, the potential dependencies between the different selected operations in the Web services are identified. There are two kinds of dependencies between operations: operation invocation dependencies and Web navigation relationships. The first kind of dependencies gives rise to Web service choreographies and the second to Web service orchestrations. These are detailed below:

5.1 Web Service Choreography Creation

In this step, we identify in the source code of the generated Web services all external calls between operations in order to replace them by Web service requests. This is explained in Algorithm 6. If the called operations are published in the same Web service of the caller operation, nothing is done, the calls are left as method invocations (see Line 6). If the called operations are present in the other published Web services these operation dependencies are replaced by Web service requests in source code of the invoking operation.

In the created graph for the Web service-oriented system, we represent each Web service request by a relationship of type WSR which relates the invoking class

node and the WSDL interface node of the invoked service (see Lines 6 to 9). An example of this relationship is given in Fig. 4, where the invocation to the *getItemByReference* method from the code of the operation *serviceTotalPriceInCart* is transformed into a WSR between the class *CartService* and the interface *wsdlProducts*. As for a local method invocation, this is represented by an ISR relationship. Fig. 4 shows an ISR relationship between the *CartService* class and the *Item* class.

Algorithm 6 Web Service Choreography Creation

```
1: procedure createChoreographies(SOS)
2:     for all op ∈ SOS do
3:         for all invOp ∈ invokedOpsFrom(op) do
4:             c1 = declaringClass(invOp)
5:             c = declaringClass(op)
6:             if c1 ∉ Cs then
7:                 wsdl1 = getServiceInterfaceOf(c1)
8:                 wsr = create WSR : wsr = (c WSR wsdl1) ∧
   wsr ∈ Rsos
9:                 createWSRequest(Code(op), invOp, wsdl1)
10:            end if
11:        end for
12:    end for
13: end procedure
```

Besides, other Web service requests can also be created starting from the parsing of client-side scripts. Indeed, the majority of modern Web applications use Ajax, which allows to build dynamic and interactive Web applications. Client-side scripts can create direct connections to the server and transfer data from clients to servers. The `XMLHttpRequest` API is the mostly used technique as an Ajax implementation [12]. In some cases, the request sent to the server asks for a server-side program which has been transformed into a Web service. We check this by the parsing of the scripts that use this API. In this case, we create a new request to this Web service. This request is added at the beginning of the source code of the Web service that is generated starting from the current Web interface. This invocation allows to update the data at the server side before it will be used by the Web service.

Let us consider our illustrative example to show how to create a composite Web service at code level. In the `Cart` Web Interface we use a client-side script to send data to the server. Before proceeding to checkout, the user can modify the quantity of the purchased items. The new quantity is sent as data to the server via a client-side script (in JavaScript using an `XMLHttpRequest` object). This client side script contains a call to a program executed at the server side corresponding to the `UpdateCart` Web interface. This program allows to update the cart and calculate the new total price. When the user clicks on the `proceed to checkout` button, a program (provided by the `Payment` Web interface) is executed at the server side. This program takes as input the new calculated total price and the new cart details.

Following our approach, the `payment` and `updateCart` operations are created respectively starting from the `Payment` and `UpdateCart` Web interfaces. So, an invocation to `updateCart` operation is added at the beginning of the `payment` operation source code. In this way, the new total price is calculated before proceeding to payment.

In addition to the invocation of the `updateCart` operation several other operation invocations are created in order to accomplish the payment process. Indeed, the `checkCreditCard` and `checkPersonalInformation` operations are invoked successively so that the input credit card information and the personal information of the user are checked. If they are valid, we call two other operations, which are `approvalPayment` operation to approve the payment and `sendEmail` operation. The `approvalPayment` operation itself calls some other operations, which are respectively: `checkCredit`, `createInvoice`, `validatePayment` and `getDeliverySchedule`. In this way, a choreography at code level is created for the operations that are involved in the payment activity.

5.2 Choreography Modeling

In this step of our approach, we reverse engineer the source code of the Web services to create high-level specifications in the BPMN language. This refers to the global view of the composition spanning multiple participants. We chose BPMN language because it provides a rich graphical notation for choreography modeling [9]. The creation of these choreographies helps in better understanding the composition of services, which for the moment can be seen at code level only.

The generation of BPMN models is based on set of rules that we have defined as mappings between the source code elements and the BPMN elements.

- For each choreography (collaboration between services and service clients) we create a BPMN model.
- Each participant (Web service or service client) in the choreography is modeled with a `Pool` element[2]. This `Pool` is used as a container of the activities that are performed by the participant. The activities rep-

[2] A Pool is the graphical representation of a Participant in a Collaboration.

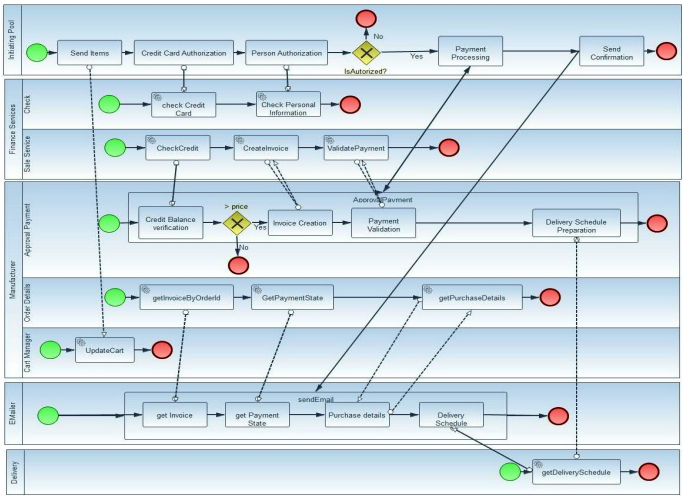

Fig. 8 A BPMN model representing a service choreography to accomplish the payment activity.

resent mainly the message exchanges of the service (or service client) with the others participants.

- If the participant is a service client, it must thus contain one or several Web service invocation(s). Each service invocation is modeled as a `Task`[3] element to be added to the created `Pool` element of the service client. This `Task` element is a kind of activities within BPMN. It is an atomic Activity within a process flow.
- The set of `Tasks` within a `Pool` are connected as a sequence (in the same order of their appearance in source code) using the `Sequence Flow` element.
- If the participant is a Web service and it has no requests for other services, the service is modeled as a `Service Task`[4] element. The `Service Task` is a kind of Task used to represent some sort of service, which can be a Web service.
- If the participant is a Web service and if it implements requests to other services, the service is modeled as a `SubProcess`[5] element. The `SubProcess` element can be a white box or a contour which shows a lower-level process that is executed by the Web service participant. Each request to an external service is identified and modeled as a `Task` element to be added as an inner activity of the `SubProcess` element. The set of `Tasks` within a `SubProcess` are connected as sequence using the `Sequence Flow` element.
- The connection between a `Task` element (which is generated for a service request) and the `Service Task` (or the `SubProcess`) which is located in a separate pool is done via a `Message Flow` element. The `Message Flow` element is used to represent the message sending or receiving between the client and the Web service participants.
- Each control statement (for example, a conditional statement) containing a service invocation is modeled with a diverging `Exclusive Gateway` (Decision) element[6]. It is used to create alternative paths within a process flow, only one of the paths can be taken. Each path is targeted to an activity element or to a default path (which can be the end of the process). The decision of the `Exclusive Gateway` can be a question or the conditional expression which is extracted from the source code (for example, the condition expression of a conditional statement). The path where the answer to this question is true is targeted to the `Task` element created for the service invocation. The two elements (`Exclusive Gateway` and `Task` elements) are connected using a `Sequence Flow` element.
- If the service invocation statements are located within a loop statement, the loop is represented with a `Loop Task` element. It has the same form as the `Task` with a loop marker in the medium. And, we move the `Task` elements created for the service invocation statements into the `Loop Task` element.

[3] A `Task` is a rounded corner rectangle which is drawn with a single thin line.

[4] A Service Task shares the same shape as the Task, which is a rectangle that has rounded corners, with a graphical marker in the upper left corner of the shape that indicates that the Task is a Service Task

[5] A `SubProcess` is an Activity whose internal details have been modeled using Activities, Gateways, Events, and Sequence Flows.

[6] It has the form of a diamond with a marker inside that is shaped like an "X"

Figure 8 shows the generated BPMN model[7], that represents the participants services which are involved in the payment activity of the example presented previously.

As we can see in this model, all the hidden choreographies are extracted and we can easily comprehend the behavior of the overall composition. This is due to our technique of reverse engineering, since only operation invocations and the control or loop statements that contain operation invocations are extracted. In this way, the developer does not need to know all the details which could be found in a large source code. Only participants (Web services) in the choreography and their exchanged messages are modeled.

By extracting this model, we believe that thanks to the high level of abstraction provided by the choreographies, evolving these applications will be simpler. Indeed, we help the developer in choosing the well suited position in the code in order to apply the necessary changes to implement an evolution scenario.

Let us suppose the following evolution scenario for the payment activity choreography. We need to add a new Web service that enhances the security of the payment process. This service can be a program that sends to the client a validation code with an SMS on her/his mobile phone. The client introduces the received code and the application checks its validity. On looking at the generated model, the developer can clearly decide to add an invocation to the new service after the `Credit Card Authorization` task in the initiating Pool.

5.3 Web Service Orchestration Creation

In this step, a set of Web service orchestrations is generated from the relationships between Web interfaces. We parse navigation documents such as JSF `faces-config` files and their navigation rules. This allows the identification of other potential collaborations of the different Web services created from these pages.

5.3.1 Navigation Rule Extraction

In some Web applications the navigation rules are not available. In this case, the hypertext links and the redirection statements/tags located in the Web application are parsed in order to create these rules. The Web application graph could be considered as a navigation model for the input Web application. We associate to each Web page node a navigation condition (NC(web page)). Indeed, the redirection statements like `response.sendRedirect("url")` are generally declared in the body of a conditional statement such as `If Statement`. Thus, depending on the condition value, the page is redirected to the appropriate destination. The used condition in this code is extracted and analyzed to be added as a condition of created navigation rule. This task requires sometimes the developer intervention to validate the generated navigation rules.

5.3.2 BPEL Process Creation Algorithm

The generated BPEL processes represent new services which implement some coarse grained functionalities provided by the application. The exchange of messages between the BPEL process and external clients (other applications or partner (Web) services) is done via a contract described in WSDL. This contract represents an interface of the BPEL composite Web service. Now, the generation of the Web service orchestration is implemented according to Algorithm 7.

In this algorithm, first all navigation paths are calculated from the Web navigation document of the parsed Web application (Line 2). Each path represents a coarse grained functionality provided to the user when (s)he navigates between the Web interfaces of this path. Therefore, for each path we create a BPEL process (Line 4) which represents a new generated Web service. After that, for each navigation rule in the current path, we identify the source operation (Line 8). The source operation corresponds to the operation that has been generated starting from the navigation's source Web interface. The same thing is done for the destination Web interface (Line 19). As specified in the algorithm, a navigation rule contains three elements: i) a source view (Line 8), which represents the Web interface(s) from which the navigation started (e.g., the Web interface presenting the form for searching items in the example introduced previously: `search.jsp`); ii) a destination view (Line 19) that corresponds to the Web interface(s) to which the user will be automatically directed (e.g., the Web interface(s) presenting the result of the search `searchResult.jsp`); and iii) an execution condition which contains an expression and a value (e.g., the expression is a call to the JavaBean method for getting the number of items found: `#{searchResult.getItemsCount}`, and the value is `"NotZero"`).

For each navigation rule, we first test if the source operation has already been called in the process while parsing another navigation rule (Line 9). This ensures that operation invocations are not duplicated. In the case of an operation which has already been invoked in

[7] The generated model is updated and validated manually by grouping the `Pools` which are of the same category and giving more readable names to `Pools`, `Lanes` (sub-partition within a Pool) and `Activities`.

Algorithm 7 WS Orchestration Creation Algorithm

```
1: procedure createBPELProcesses(naviRules)
2:     naviPaths = calcNaviPaths(naviRules)
3:     for all path ∈ naviPaths do
4:         process = ProcessFactory.newInstance()
5:         seq = process.createSequence()
6:         returnedVal1 = process.createVariable()
7:         for all navigRule ∈ path do
8:             opFrom = parseSourceView(navigRule.sView)
9:             if !(opFrom isPreviouslyInvokedIn process) then
10:                op1 = process.createInvocationTo(opFrom)
11:                op1.setParameters(variablesofprocess)
12:                returnedVal1 = op1.invoke()
13:                process.store(opFrom, returnedVal1)
14:                seq.add(op1)
15:            else
16:                returnedVal1 = getStoredReturnedValBy(opFrom)
17:            end if
18:            ifActivity = parseConditionExpression(navigRule, process, seq)
19:            opTo = parseDestinationView(navigRule.dView)
20:            if !(opTo isPreviouslyInvokedIn process) then
21:                op2 = process.createInvocationTo(opTo)
22:                matchedParts = calculateSimilarity(opFrom.outputMsg, opTo.InputMsg)
23:                op2.setParameters(variablesinprocess + returnedVal1, matchedParts)
24:                returnedVal2 = op2.invoke()
25:                process.store(opTo, returnedVal2)
26:                ifActivity.add(opTo)
27:            else
28:                seq = addPickToProcess(navigRule, process, seq)
29:            end if
30:        end for
31:    end for
32: end procedure
```

the process, we just get the returned value (Line 16). This kind of values are stored after each operation invocation (see Lines 13 and 25). Then, we parse the condition part in the navigation rule, by calling a function (see Line 18). In this function, we get the expression of the condition, which corresponds to the operation to be invoked. We test if this operation is not invoked previously in the process. In this case, we create in the process an invocation to this operation. After that, we create in the process an `If Activity`. It is used to compare the obtained value after invoking the operation defined in the condition's expression and the condition's value. If the two values are equal, then we invoke the corresponding destination operation (see Lines 19 to 24). We need to test before, if the destination operation has already been called in the process while parsing another navigation rule. This means that there is a cycle in this navigation path. The cycles occurs when the user wants to navigate again in a path with new input values. So, all invoked operations in this cycle can be invoked again with the new input values. We deal with cycles by calling of the function `addPickToProcess` (see Line 28). Fig. 9 shows an excerpt of an abstract process, which represents the sequence of generated activities to be added in the process after calling the `addPickToProcess` function. This abstract process contains mainly the activities: `Pick` and `RepeatUntil`(Loop).

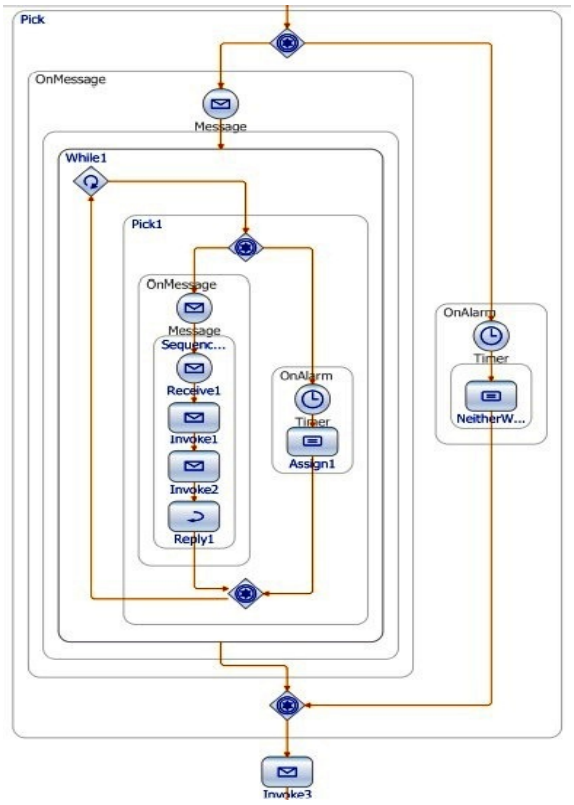

Fig. 9 An excerpt of a BPEL process representing the created activities to deal with a cycle.

The loop activity is used to repeat the set of created invocations starting from the cycle. The pick activity allows the process to block and wait for one or a set of suitable message(s). The arrival of a message indicates that the user needs to repeat the invocation of operations in the cycle. When one of these messages is received, the associated activity is performed and the pick completes. If none of these expected messages is received within a certain period of time [8], the pick can specify an exceptional behavior to be performed (in our algorithm, an `OnAlarm activity` is added, which allows to the process to wait). In this way, the cycle of the navigation path is considered in the BPEL process.

[8] We give a default value for this time interval, which could be modified by the developer on the generated BPEL process

In orchestration creation, before each operation invocation (see Line (24) in the algorithm above), we prepare the list of arguments. A matching of the variables' names in the orchestration and the arguments of the operation to be invoked is performed. In this way, we check the syntactic composability [25] of the two operations to be composed and we ensure that arguments are passed in the correct order (see Lines (22) and (23)).

Each generated composite service is modeled as a sub-graph that belongs to the graph which represents the migrated service-oriented system. In this sub-graph we create a WSDL interface node and BP(bpel) node and we connect them via an IIR relationship. These nodes represent respectively the interface and implementation of the generated composite service. The invocations to operations which belong to other Web services are represented with WSR relationships. These relationships relate the BP(bpel) node of the generated composite service and the WSDL interface node of each invoked service.

5.3.3 Example of BPEL Process generation

Let us take the example presented in Section 2. A BPEL process is generated for the navigation path which represents a successful purchasing. The process first invokes the `basicSearch` operation of the first service. Then, it stores the result into a variable and invokes `getItemsCount` operation of the same service to get the number of found items. If the returned value is equal to `"NotZero"`, the `addItems` operation of the `CartService` is invoked. The selected items (received using a `Receive activity`) by the customer are the input of this operation. The returned total price is stored. After receiving the email and the password of the customer the `signIn` operation is invoked. If the identification is passed successfully, an invocation to the `DeliveryService` is done. After that, the stored total price is used as input for invoking the `PaymentService`. At last, the `sendMail` operation is invoked with the necessary data.

6 Experimentation: A Case study

As stated at the beginning of the paper, when developers want to implement extensions to Web applications, without using our approach, they can either: i) develop these extensions from scratch by writing programs that send HTTP requests to the Web applications and then analyze the returned HTTP responses; or ii) create "manually" Web services that publish the functionality of the Web application[9] and then write programs that invoke these services. We have aforementioned that this task of developing extensions is costly: cumbersome and so time-consuming.

In order to show that our approach reduces the cost of the development of extensions to Web applications, we have conducted an experiment on three real-world Web applications of different sizes. In this experiment we have in particular addressed the following research questions:

- **RQ1**: What is the performance of the process of identification of operations and Web services that are generated and published?
- **RQ2**: What is the additional cost induced by the proposed approach?

For answering the first research question, we measured in our experiment the performance of our approach taking the definition of "performance" from the information retrieval domain. We measured thus the precision and the recall on the results of the steps in the process: i) the identification of published operations, ii) and the creation of BPEL processes.

For answering the second research question, we have measured: i) the size (in terms of number of statements) of the same extensions developed first without our approach, and then with our approach; ii) the time taken in the development of many extensions to see at what time (from how many developed extensions) we can see the benefits of using our approach (initial cost of Web service generation amortized).

The chosen applications for our experiment are: (i) an E-Auction application [15] that provides via its Web interfaces the functionality for buying and selling second-hand goods by bidders and sellers[10]. (ii) An Online Music Portal, which is a JEE Web application[11]. This application offers to users and administrators several functionalities such as: searching, purchasing and managing songs. (iii) A simulated version of a Web service search engine (Seekda). This application provides a set of functionalities, such as, searching for public Web services in the Internet. Table 2 describes the size of the three Web applications, where `NSAC` represents the Number of Server-side scripts And Classes. `NLOC` represents the Number of Lines Of Code in the Web application. These Web applications are considered as

[9] This task is not really fully manual. We consider the use of tools for annotating code and then generating Web services from this annotated code.

[10] It has been downloaded from the following GitHub repository: https://github.com/FrancescaRodricks/E-Auction-SE-Project

[11] Downloaded from: https://github.com/sahebkanodia/onlinemusicportal

Table 2 Size of the three Web applications

Systems	NSAC	NLOC
E-Auction Application	31	1600
Music Portal Application	32	570
Seekda Application	10	850

inputs for our tool (WSGen: Web Service Generator). We have presented in our previous work [39] what are the generated Web services and their compositions obtained from the Seekda Web application.

6.1 Experiment for RQ1

In order to answer the first research question, we have involved in our experiment four PhD students mastering Java EE. First, each participant is asked to annotate the code of the three Web applications using EJB 3 annotations. Then, we have used Eclipse JEE to generate Web services starting from the annotated classes and methods of these applications. This is what we consider the "manual" Web service generation. We have then measured the number of the generated operations. In addition, we have asked these PhD students to imagine all the possible pertinent operations that can be created from each application's source code. We have calculated the number of these operations. After that, in order to calculate Recall and Precision for the operation identification step, we have measured for each application:

- **True Positives (TP)**: the operations identified and published by WSGen and which are also created manually.
- **False Positives (FP)**: the operations created by WSGen, but which are eliminated manually.
- **False Negatives (FN)**: the operations created manually without our approach and which have not been generated by WSGen.
- **True Negatives (TN)**: the operations eliminated by WSGen and are not created manually.

The Precision is the ratio of the number of true positives to the total number of all created operations by WSGen ($TP + FP$). **Precision** $= \frac{TP}{(TP+FP)}$.

The Recall is the ratio of the number of true positives to the number of operations that should be published ($TP + FN$). **Recall** $= \frac{TP}{(TP+FN)}$.

Table 3 depicts the obtained values of precision and recall for the three applications. The obtained values show that precision of the operations identification is relatively good except the case of the Music Portal application where the precision is low. In this case, the elimination of the operations provided to the administrator of the application (adding users, ...) cannot effectively be automated and requires the developer knowledge. By this intervention of the developer the value of FP becomes high, which affects negatively the value of the precision.

Besides, in the third column of Table 3 (TP) we can see a little difference between the values obtained by the involved developers. This is due to the fact that there are some correct operations which are identified by some developers but they are not identified by the others. This variance is due in general to several factors such as: the level of developer skills, the time spent in analyzing the code (careful code review or not), the complexity and the size of the code and the availability of Web application documentation and architecture.

However, the measures show that the recall rate is relatively high for the three applications. This means that the correctness level of our approach is relatively good (most of the identified operations by WSGen are also created manually). Nevertheless, additional operations are created manually and cannot be identified automatically by WSGen (False Negatives). An example of these operations for the E-Auction application is `getAuctionListForProduct` operation. It returns a list of created auctions for a specific kind of product. This functionality was not provided directly by the E-Auction application, but it was easy to create it by invoking an existing method in the Web application and then filtering its returned results that correspond to a specific product. Additional operations are created manually (False Negatives) by slicing the source code of some programs in the Web applications. The creation of this kind of operations implies complex human thinking and this cannot be fully automated.

There are many variables which are involved in the service and operation identification which made this task too complicated and time-consuming. In such a scenario, identification of candidate services and operations within a large source code is challenging. Various strategies and ways could be adopted such as: using architectural reconstruction approaches [**?**], pattern detection [**?**] or concept analysis and program slicing techniques [**?**].

After the evaluation of performance for the operation's identification step, now we have made measures of performance in order to calculate the recall and the precision for the step of the generation of BPEL processes. We have asked the four PhD students to extract all possible combinations that could be considered as pertinent Web service orchestrations. Then, these have been compared with the generated orchestrations by WSGen. For these measures, the **True Positives**

Table 3 Recall and Precision calculation for the operation identification step

Involved developers	Systems	TP	FP	FN	Precision	Recall
PhD Student 1	E-Auction Application	25	9	3	0.73	0.89
	Music Portal Application	10	15	2	0.40	0.83
	Seekda Application	12	2	1	0.85	0.92
PhD Student 2	E-Auction Application	28	9	7	0.75	0.80
	Music Portal Application	12	15	3	0.44	0.80
	Seekda Application	12	2	2	0.85	0.85
PhD Student 3	E-Auction Application	26	9	4	0.74	0.86
	Music Portal Application	13	15	5	0.46	0.72
	Seekda Application	9	2	2	0.81	0.81
PhD Student 4	E-Auction Application	27	9	8	0.75	0.77
	Music Portal Application	14	15	6	0.48	0.70
	Seekda Application	11	2	3	0.84	0.78

Table 4 Recall and Precision calculation for the step of BPEL generation step

Involved developers	Systems	TP	FP	FN	Precision	Recall
PhD Student 1	E-Auction Application	32	4	10	0.88	0.76
	Music Portal Application	10	2	2	0.83	0.83
	Seekda Application	10	2	1	0.83	0.90
PhD Student 2	E-Auction Application	37	4	15	0.90	0.71
	Music Portal Application	11	2	2	0.84	0.84
	Seekda Application	10	2	2	0.83	0.83
PhD Student 3	E-Auction Application	29	4	9	0.87	0.76
	Music Portal Application	13	2	2	0.86	0.86
	Seekda Application	8	2	1	0.80	0.88
PhD Student 4	E-Auction Application	35	4	13	0.89	0.72
	Music Portal Application	15	2	2	0.88	0.88
	Seekda Application	12	2	3	0.85	0.80

are the number of BPEL processes which are created manually and with WSGen. The **False Positives** are the BPEL processes created by WSGen and eliminated manually. The **False Negatives** are the processes created manually but which have not been generated by WSGen.

As we have seen in the first evaluation, there are some operations which have been created manually and which are not generated by WSGen. As a result, we cannot rely on the previous operation sets to measure recall and precision for BPEL process generation. To deal with this issue, we have decided to consider in our measurement only the operations that are correct (created both by WSGen and manually). After that, we measured the number of BPEL processes that invoke these operations. The obtained values are depicted in Table 4. We can see in this table that the precision is high for the three applications, which demonstrates that the step of orchestration generation gives good results with a minimum rate of errors.

Besides, we can observe also in Table 4 that the recall is relatively high for the three applications. Despite that, there is a number of orchestrations which are created manually and which are not generated by WSGen. This is due to the use of navigation models for generating BPEL processes. As result, some combinations of operations cannot be identified from these models but they are defined manually (False Negatives). An example of these combinations is an orchestration that invokes operations which are created starting from programs in back-end components of Web applications. This kind of operations are not always invoked directly from BPEL processes that are generated by WSGen. This fact does not have an influence on the correctness of our approach, because most of the orchestrations which represent the functionality used by end-users via Web interfaces are created by WSGen.

6.2 Experiment for RQ2

To answer RQ2, we estimated the developer's effort when (s)he implements extensions to the three Web applications with and without our approach. This part of the experiment is based on the following steps:

– We have implemented extensions to the three Web applications without using our approach and then the same extensions have been developed using our approach. In the extensions, which have been developed without using our approach, we have written

Java programs that send HTTP requests to the running Web applications and by analyze the HTTP responses. The implemented extensions are:
  - For the E-Auction: the extension allows to the user to create alerts by sending an email when particular products with specific features are offered by the E-Auction application.
  - For the Music Portal: the extension is a mobile front-end for this application, for accessing the searching functionality of the application from an Android tablet.
  - For Seekda: the extension searches for Web service descriptions from the search engine by sending keywords and then retrieves WSDL files from the obtained results (in order to exploit them to make classifications of Web services [2]).
- We evaluated the number of statements in each implemented extension to each Web application and we compare the obtained values for the two approaches. NS_{WSGen} is the number of statements for implementing an extension by invoking the generated Web services. NS_{HTTP} is the number of statements for implementing an extension by sending HTTP requests to the Web application and analyzing the returned responses.
- We calculated *StmRatio* which is the ratio between NS_{WSGen} and NS_{HTTP} for the three Web applications ($StmRatio = NS_{HTTP}$ / NS_{WSGen}).

The values for *StmRatio* for the three applications are: 3.76 for the E-Auction application, 3.6 for the Music Portal application, and 3.2 for Seekda. All the values confirm that the extensions developed using our approach are more than three times smaller than the same extensions developed using HTTP requests (answer to RQ2).

We can observe that the comparison made, between the two ways of developing extensions, to answer the second research question does not take into consideration the initial overhead of our approach. This overhead can be quantified by measuring the number of OCL expressions and the number of interactions with WSGen GUI (number of clicks in order to validate operations, for example). But this cannot be added up to the number of statements and then be compared with the number of statements when developing the extensions without our approach. Then, we have decided to compare the total time spent during the development of several extensions to one of the three Web applications, which is the Music Portal application. In addition to the firstly created extension we have implemented three other simple extensions to this Web application. Then, in order to not limit ourselves to four extensions of this Web application, we have decided to create several fictive (simulated) extensions. For each simulated extension we vary randomly the value of NS_{HTTP} and we calculate NS_{WSGen} which is equal to: $NS_{HTTP} \times StmRatio$.

Table 5 Time for the four extensions of the Music Portal application

Extensions	T_{HTTP}	T_{WS}
Extension 1	1.94	1.9
Extension 2	1.2	0.85
Extension 3	0.94	0.77
Extension 4	0.92	0.75

The comparison now is based on the calculation of the following values:

- T: represents the total time for generating primitive and composite Web services from the Web application using our approach. This time is the sum of the global time for executing our tool and the time for the intervention of the developer: writing OCL constraints and validating the generated operations and services.
- T_{WSG}: represents the time spent during the implementation of an extension to a Web application by invoking the generated Web services. This time is the sum of the global time needed for a developer to understand the service descriptions and/or the BPEL processes in addition to the time for programming and testing an extension.
- T_{HTTPG}: represents the needed time to implement the same extension without using our approach. This time is the sum of the estimated time for navigating between the interfaces of the Web application in order to learn and save, before the development of the extension, the content of the HTTP requests (and their responses) sent to (resp. received from) the Web application and the global time to implement and test this extension.
- T_{WS}: is the average time for implementing one statement within an extension using our approach. This time is equal to T_{WSG}/NS_{WSGen}.
- T_{HTTP}: is the average time for implementing one statement within an extension without using our approach. This time is equal to T_{HTTPG}/NS_{HTTP}.

Table 5 shows the average values of time for implementing one statement in the four extensions of the Music Portal Web application. We can observe that the values of T_{HTTP} and T_{WS} decrease from one extension to the next one. This is related to the fact that the developer acquires a programming experience during the implementation of an extension, which allows her(him) to implement the next extension in less time. This experience is affected by several factors such as the use of the same API. In this experimentation, we have

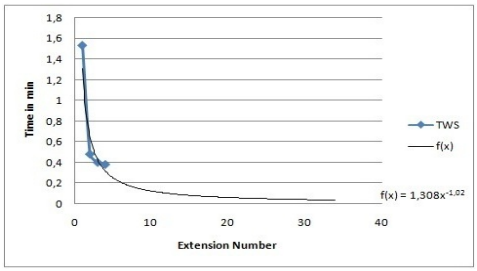

Fig. 10 Result of the application of the regression technique for our approach.

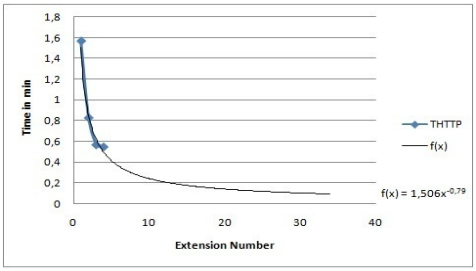

Fig. 11 Result of the application of the regression technique for the HTTP approach.

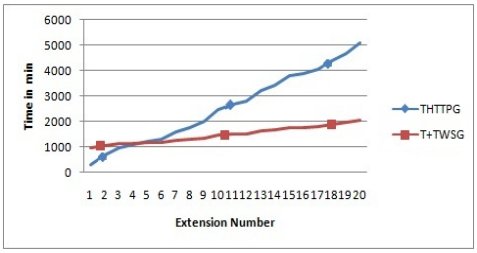

Fig. 12 Variation of the time over the number of extensions in the Music Portal Application

supposed that all the simulated extensions are implemented using the same API and in the same programming language. Besides, we have estimated the time for rewriting the code of the four extensions (as if it was a "mechanical" task that does not imply thinking). The estimated value of T_{HTTP} and T_{WS} is equal to 0.37 min. This is the minimal time. Now, we would like to predict the values of T_{HTTP} and T_{WS} for each simulated extension. For this, starting from the values in Table 5 we have used a regression technique to obtain two curves of trends and their equations. These two equations allowed us to extrapolate the values of T_{HTTP} and T_{WS} for the simulated extensions. As we have aforementioned, the values in Table 5 have a decreasing trend that tends to the value 0.37. So, to determine the curves of trends we used an inverse power regression model, which is defined mathematically as following: $f(x) = a \times x^b$, where a and b are constants and b is negative. $f(x)$ represents the value of T_{HTTP} or T_{WS} for the extension number: x. It is clear that this function converges to zero. But we have mentioned that the minimal value of time for T_{HTTP} and T_{WS} is 0.37. Thus, the function that we need in our calculation could be defined as the following: $g(x) = f(x) + 0.37$. To calculate f(x), we have first subtracted the value 0.37 from the values in Table 5. After that, we have used these new values as input when we have applied the regression technique. The obtained curves of trends and their equations[12] are shown in Fig 10 and Fig 11. Now, in order to estimate the global time(T_{HTTPG} and T_{WSG}) for implementing each simulated extension we use a function h which is defined mathematically as the following: $h(x, r) = g(x) \times r$, where r represents the number of statements NS_{HTTP}[13] or NS_{WSGen}[14] in each simulated extension. Thus, the result of the calculation of time for implementing all the extensions of the Music Portal are shown in Fig12.

[12] These curves and equations are generated using the tool provided by the Microsoft Office Excel 2007.

[13] NS_{HTTP} is generated randomly.

[14] NS_{WSGen} is calculated by: $NS_{HTTP} \times StmRatio$.

We have accumulated the global time for each simulated extension (the time to develop the whole extension, not the time for writing one statement). We can observe in the curves that the time for implementing the first extension using our approach is relatively high comparatively to the time spent during the implementation of the same extension without our approach. This time is equal to $T + T_{WSG}$, which represents the initial overhead of our approach (which is considered only one time) and the time for implementing the first extension. We can see in Fig. 12, starting from the fourth extension (it represents the point where the two curves intersect), that our method becomes beneficial and the global time for implementing and testing extensions becomes less than the global time for doing the same thing without our approach. In addition, we can observe that the slopes of the two curves are different. The curve related to the development of extensions without our approach grows faster. This shows the amortization of the initial overhead of our approach over the development of multiple extensions (answer to the second part of RQ2).

6.3 Discussion & Threats To Validity

For the external validity, and in order to generalize the experiment results of the case study we need to take into consideration several aspects such as, the type and the size of the selected Web applications. We conducted our experiment on different types of Web applications that have relatively medium sizes. To achieve results which

can be confidently generalized we need to run this experiment on larger Web applications developed using different technologies (languages and/or frameworks). But the fact that our applications have medium sizes helped us in manually creating Web services in a reasonable time. In addition, the hypothesis about the use of HTTP requests (using java.net API of the standard library of the JDK) for developing extensions slightly biases the experiment. Indeed, this is not the most effective way of doing, but we have considered it because it is the most straightforward and the standard way for developing Java-based extensions of Web applications (for Web scraping, for instance). Besides, this is the only way of developing such extensions for Web applications that do not provide a service-oriented REST (or SOAP-based) API, which is the case of most Web applications today.

On what concerns the internal validity, we can be tempted to say that, as the authors have been involved (not in the manual annotation of the Web application code to generate Web services in the first part of the experiment, but) in the development of the extensions without WSGen in the second part of the experiment, the results would be biased. This fact does not have an influence on the experiment results. Indeed, the way of developing the extensions by using HTTP requests is greatly different from using Web services. The fact that the developers know what would be the generated Web service interfaces of the Web applications does not impact their way of programming the HTTP requests using the java.net API.

7 Related Work

We have grouped the existing approaches and tools for migrating systems to (Web) services-oriented applications in the following categories:

7.1 Approaches for generating Web services from software components

The proposed approach in [20] aims to convert software components into Web services. The components are saved by the provider in a component repository. The client specifies a request for searching a given functionality in components which are implemented in C++ or Java. A set of services are generated automatically for the desired functionalities and returned to the client. A. Marinho *et al.* [24] propose a similar approach to [20]. In addition, the proposal allows the generation of services starting from components, which are written in different programming languages. In our approach we do not react to a client request, but we propose to the developers of the Web application to anticipate the export of some functionalities as Web services. In this way third party developers can make remote extensions of the services exposed by the interfaces of the Web components. However, in [20,24], only business functionalities implemented in software components are transformed. In our approach, the Web interfaces, the business functionalities and the navigation between Web interfaces are converted into stateless Web services and compositions of them. In [11], the authors present a conceptual model of Web components. They propose a method for composing a set of services provided by Web components using parameterized contracts. These contracts link the services in the provided interfaces of a given component to the services of its required ones. To satisfy a given functionality when a composition is under construction, a service is included if all its required services are satisfied by the component's environment. If some required services are not satisfied, other provided services from other components are integrated. In our approach, the Web components that we deal with do not define required interfaces. They refer to industrial solutions of Web development (like Java EE). In addition, we build compositions of services as BPEL processes starting from existing Web and business logic code, while in [11], compositions are built starting from formal definitions (contracts) associated to some candidate services in a repository.

7.2 Approaches for migrating Web applications to SOA

Some works have been proposed recently for migrating Web applications to service oriented architectures (SOA), such as [1,38,41,17]. The authors of [1] propose a semi-automatic transformation process to migrate Legacy Web applications (implemented using the PHP scripting language) to a service oriented application. In contrast to our approach, the authors of [1] focus on the service migration aspect, while the identification of services is done manually based on the developer knowledge. In our work, we deal with this aspect in an automatic manner. We parse automatically programs executed at sever side in the aim of identifying the potential operations to be published as Web services. The proposed work in [38] addresses the problem of extracting Web services from Web applications. This work proposes a model for decomposing and abstracting a Web application into modular building blocks forming the desired Web services. The decomposition model is created by modeling each human transition

from page to page as modular pieces of the entire service. The abstract model consists of argument passing, data extraction and context propagation in each transition from page to page. Using these two models and a set of configuration files created manually by the developer, Web service wrappers can be created for the Web applications. The decomposition technique in [38] do not work *a priori* with Web applications that use techniques for retrieving remote data asynchronously (like AJAX). In our approach, we deal with this kind of applications by transforming the scripts at client-side into Web service requests, while the executed program at server side is exported as a Web service. In addition, the page transition approach in [38] is applied to traditional Web pages, where the navigation is done with hyperlinks contained in pages. Actually, modern Web applications that use navigation documents to implement dynamic transitions from page to page (such as in JSF Framework) are not addressed. In the proposed approach in [41], RESTful services can be extracted from Web applications. The approach is based on the capturing of scenarios executed by a user for the task to be migrated as a RESTful service. The input, output and HTTP methods of the task are identified by the analysis of the annotation logs and the execution logs. In another work, Wike [17] generates virtual Web services by extracting information from Web pages. Users can define patterns which are used to extract partial information from Web pages. The extraction function can be used to generate a Web service that returns the result of the extraction process. Content-based Web pages are not the main concern in our approach. Indeed in our process, Web components including Web interfaces and business logic implementation are the artifacts concerned by Web service generation. These works are complementary solutions to our work. Web services that are generated using our approach starting from Web components, which produce to users during execution a large quantity of content, can be enhanced with new operations that return only partial information (texts, images, ...) using Wike. Invocations to these new operations can be added to the orchestrations generated by WSGen. Another work is proposed by [23] relies on the analysis of the client-side Web application code. The authors consider several behaviors which could be reused in a large number of Web applications [22]. They propose an approach to identify and extract the code which implements certain behaviors. The proposal is based on dynamic analysis, which relies to the execution of scenarios and saving the executed code (client-side code) responsible for an expected behavior. In addition to the behavior identification, the approach can extract library functionalities and identify (or delete) the code that does not implement any behavior (improve the performance). This work is complementary to our proposal as it deals with client side scripts' code. Analyzing such kind of client code is one of the perspectives of our work. For the moment, we partially deal with it in choreography creation when Ajax is used in Web application.

7.3 Approaches for migrating legacy systems to SOA

Several techniques and methods have been proposed to face the problem of migrating legacy systems to service-oriented architectures (SOA). The works in [21,37,44] are representative examples of such approaches. The SMART approach [21] aims at assisting organizations to migrate their legacy systems to SOA in a systematic way. The legacy systems functionalities, or subsets of them are exposed as services. The proposal is based on an interview guide, which is presented to the developer in terms of questions. These questions concern issues about the process of the migration. Based on developer's responses, the degree of the difficulty and the required effort to make such migration are determined. Sneed *et al.* [37] present a set of metrics to be considered in the identification of services in legacy systems. The identified legacy code will be wrapped and make it available as a web service. Zhang and Yang [44] propose a re-engineering approach based on clustering to integrate legacy systems in SOA. The extracted functional legacy code is restructured to facilitate the Web service construction. Another solution has been proposed in [8] to migrate form-based legacy systems into Web services based on a wrapping approach. In the form-based legacy system the flow of data between the system and the user is described by a sequence of query and response interactions, which is converted into message requests from the client and message responses from the service provider. In this approach, the behaviors accrued when the user interacts with the legacy system is modeled in term of finite state automata, based on a black box reverse engineering technique. This specification will be interpreted by the wrapper. Comparing these works to our approach, we do not create wrappers for the functionalities exposed by the Web application, but we create a new Web service application starting from the functionality exposed by the Web application. The generated Web service application will be eventually used and extended remotely by third party developers. However, the existing Web applications are kept running and accessible for end users.

7.4 Model-Driven Approaches for generating Web service-oriented applications

Many works in the literature propose model-driven techniques to generate Web service-oriented applications. The authors of [4] propose an approach based on MDA[15] to transform the UML2 sequence diagrams to BPEL processes. This approach aims in particular to assist the developer in coding BPEL specifications. [16] proposed a model-driven process for building Web service compositions. The WSDL descriptions are transformed into UML models. These models are integrated by the developer to form composite Web services, which contain interface and workflow descriptions. Interface models are described using stereotyped UML class diagrams and workflow models are represented by stereotyped activity diagrams. At the end, a set of WSDL descriptions are generated for the resulting composite services. This work provides means for making forward engineering (UML to WSDL and BPEL) and reverse engineering (WSDL to UML) by specifying bidirectional transformation rules. In [42], the authors propose transformation rules for converting orchestration models specified in CCA (Component Collaboration Architecture), which is part of the UML profile for Enterprise Distributed Object Computing (EDOC [29]), into BPEL specifications. Another model-driven approach for creating service-oriented solutions has been proposed in [19]. In this work, a UML profile has been defined for service-oriented applications.

All these works are complementary to our approach. In our work the transformations are made from PSM to PSM. Web components, which are models specific to a given platform (in the current implementation, Java EE), are converted into Web services, which are considered as another platform-specific model (WSDL, Java and BPEL, in the actual version of WSGen). The UML profile presented in [19] can be used to define high-level models of the generated Web services. The other approaches can be used to make a reverse engineering of the generated Web services or orchestrations and obtain more understandable models (compared to code). In addition, all these related works focus on UML modeling and generating new Web services starting from models of a high level of abstraction. In our approach, we worked on the transformation of existing Web code.

7.5 Approaches for Web Service Composition

A summary of proposed solutions, standards and Frameworks for Web service composition is presented in [26]. A survey of existing methods and approaches for reliable composite services is presented in [18]. Several other automatic, semi-automatic and manual service composition approaches are proposed in the literature such as [34, 28, 36, 43, 35, 25, 5]. Paik et al. [34] propose a nested multilevel dynamic composition model which provides functional scalability and seamless composition. Oh et al. [28] consider the automatic composition of Web services as AI planning and network optimization problems. In [36] a context-based semantic approach is proposed for classifying and ranking Web services in order to compose them. The classification is based on the analysis of the WSDL documents and free text descriptions of the Web services. Medjahed et al. [25] propose a composability model to check whether Web services can be composed without failure during their execution. In this model, the Web services are compared through four levels: syntactic, static and dynamic semantic and qualitative levels. All these works deal with complementary aspects to our proposal. We create automatically compositions of the services generated from Web applications. In our work, we have used the composability model of [25] to check the syntactic composability of the services. Semantic composability is one of the perspectives of our work.

8 Conclusion and Future Work

Nowadays, there is a real need for shifting from Web applications targeting exclusively humans into Web service-oriented applications. Examples of scenarios where this need is felt, for example, when we want to build mobile applications by using data from existing Web applications or when we want to implement a new "niche" business logic in the Web, underlying a large/-famous Web application. After this shift, the obtained result will thus enable third tier developers to build systems by reusing services provided by the existing Web applications instead of creating them from scratch or dealing with complex HTTP interactions. In this paper, we have presented an approach that helps developers to create service-oriented applications starting from their Web applications. In addition to create individual Web services from Web interfaces and existing primitive functionality, we also allow to generate composite Web services by assembling the created individual Web services as coarse-grained functionalities. All these "emerging Web services" contribute in opening Web applications for third-tier extension development.

In the near future we plan to extend the proposed method by implementing more sophisticated techniques for grouping complementary operations in Web services, based on "text-mining" of Web components' doc-

[15] OMG's Website: http://www.omg.org/mda/specs.htm

umentation. At the conceptual level, we plan to study the formalization of the performed transformation as a set of high-level declarative rules. We then define such rules in a QVT-compliant language [31] and thus integrate our solution in a Model-Driven Engineering process. Furthermore, we plan to address more accurately the security issues when migrating Web application toward Web service oriented systems. In fact, in the current work, we deal with Web interfaces that use a secure protocol such as TSL (SSL). Actually, Web applications start using a third party secure delegation service (such as OAuth authorization [**?**]) to enhance their access security. As a future work, we intend to study the migration of such kind of Web applications towards Web service-oriented systems that use this secure delegation service.

References

1. Asil A. Almonaies, Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. A Framework for Migrating Web Applications to Web Services. In *Proc. of ICWE*, 2013.
2. Z. Azmeh, M. Driss, F. Hamoui, M. Huchard, Moha N., and C. Tibermacine. Selection of composable web services driven by user requirements. In *Proc. of IEEE ICWS*, 2011.
3. Linda Badri and Mourad Badri. A proposal of a new class cohesion criterion: An empirical study. *Journal of Object Technology*, 3(4):145–159, 2004.
4. Bernhard Bauer and Jorg P. Muller. Mda applied: From sequence diagrams to web service choreography. In *Proc. of ICWE*, 2004.
5. Amel Boustil, Ramdane Maamri, and Zaidi Sahnoun. A semantic selection approach for composite web services using owl-dl and rules. *Serv. Oriented Comput. Appl.*, 8(3):221–238, September 2014.
6. L.C. Briand, Y. Labiche, M. Di Penta, and H. Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE TSE*, 31:833–849, 2005.
7. Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.
8. Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *JSS*, 81(4):463–480, 2008.
9. Gero Decker, Oliver Kopp, Frank Leymann, Kerstin Pfitzner, and Mathias Weske. Modeling service choreographies using bpmn and bpel4chor. In *Proc. of CAiSE*, 2008.
10. T. U. Dresden. Ocl compiler web site. http://dresden-ocl.sourceforge.net/, 2009.
11. Yui-Ku Fei and Zhijian Wang. A concept model of web components. In *Proc. of IEEE SCC*, 2004.
12. David Flanagan. *JavaScript - The Definitive Guide (6th ed.)*. O'Reilly, 2011.
13. Eclipse Foundation. Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/?project=emf, 2009.
14. Eclipse Foundation. Model Development Tools website. http://www.eclipse.org/modeling/mdt/, 2009.
15. Rodricks Francesca, Chauhan Sunil, Pascoala D'Souza, Kumar Subodh, and Fernandes Leanne. E-Auction System, project of Goa University . https://github.com/FrancescaRodricks/E-Auction-SE-Project.
16. R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web service development. *IJWSR*, 1(4):1–13, 2004.
17. Hao Han and Takehiro Tokuda. Wike: A web information/knowledge extraction system for web service generation. In *Proc. of ICWE*, 2008.
18. Anne Immonen and Daniel Pakkala. A survey of methods and approaches for reliable dynamic service compositions. *Serv. Oriented Comput. Appl.*, 8(2):129–158, 2014.
19. Simon K. Johnston and Alan W. Brown. A model-driven development approach to creating service-oriented solutions. In *Proc. of ICSOC*, 2006.
20. Roger Y. Lee, Ashok K. Harikumar, Chia-Chu Chiang, Hae Sool Yang, Haeng-Kon Kim, and Byeongdo Kang. A framework for dynamically converting components to web services. In *Proc. of SERA*, 2005.
21. Grace Lewis, Edwin J. Morris, and Dennis Smith. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. In *Proc. of CSMR*, 2006.
22. Josip Maras, Jan Carlson, and Ivica Crnkovi. Extracting client-side web application code. In *Proc. of WWW*, 2012.
23. Josip Maras, Maja Stula, Jan Carlson, and Ivica Crnkovic. Identifying code of individual features in client-side web applications. *IEEE TSE*, 39(12):1680–1697, 2013.
24. Anderson Marinho, Leonardo Gresta Paulino Murta, and Cludia Werner. Extending a software component repository to provide services. In *Proc. of ICSR*, 2009.
25. Brahim Medjahed and Athman Bouguettaya. A multilevel composability model for semantic web services. *IEEE TKDE*, 17(7):954–968, July 2005.
26. Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Comp.*, 8:51–59, 2004.
27. OASIS. Web Services BPEL Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.
28. Seog-Chan Oh, Dongwon Lee, and Soundar R. T. Kumara. Effective web service composition in diverse and large-scale service networks. *IEEE Trans. Serv. Comput.*, 1(1):15–32, 2008.
29. OMG. UML Profile for Enterprise Distributed Object Comp. http://www.omg.org/technology/documents/formal/edoc.htm.
30. OMG. Object Constraint Language specification, version 2.0, document formal/2006-05-01. http://www.omg.org/spec/OCL/2.0/, 2006.
31. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/, 2008.
32. OMG. Business Process Model and Notation (BPMN) Version 2.0. http://www.omg.org/spec/BPMN/2.0/, 2011.
33. OMG. Unified Modeling Language (UML) Superstructure specification, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/, 2011.
34. Incheon Paik, Wuhui Chen, and Michael N. Huhns. A scalable architecture for automatic service composition. *IEEE Trans. Serv. Comput.*, 7(1):82–95, 2014.

35. Kaijun Ren, Nong Xiao, and Jinjun Chen. Building quick service query list using wordnet and multiple heterogeneous ontologies toward more realistic service composition. *IEEE T. Services Computing*, 4(3):216–229, 2011.
36. Aviv Segev and Eran Toch. Context-based matching and ranking of web services for composition. *IEEE Trans. Serv. Comput.*, 2(3):210–222, 2009.
37. Harry M. Sneed. Integrating legacy software into a service oriented architecture. In *Proc. of CSMR*, 2006.
38. Michiaki Tatsubori and Kenichi Takashi. Decomposition and abstraction of web applications for web service extraction and composition. In *Proc. of the IEEE ICWS*, 2006.
39. Chouki Tibermacine and Mohamed Lamine Kerdoudi. Migrating component-based web applications to web services: Towards considering a "web interface as a service". In *Proc. of IEEE ICWS*, 2012.
40. Okba Tibermacine, Chouki Tibermacine, and Foudil Cherif. Wssim: a tool for the measurement of web service interface similarity. In *Proc. of CAL*, Toulouse, France, 2013.
41. Bipin Upadhyaya, Foutse Khomh, and Ying Zou. Extracting restful services from web applications. In *Proc. of IEEE SOCA*, 2012.
42. X. Yu, Y. Zhang, T. Zhang, L. Wang, J. Zhao, G. Zheng, and X. Li. Towards a model driven approach to automatic bpel generation. In *Proc. of ECMFA*, 2007.
43. Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.
44. Z. Zhang and H. Yang. Incubating services in legacy systems for architectural migration. In *Proc. of APSEC*, 2004.