Journal of Web Engineering, Vol. 12, No. 1&2 (2013) 131–158
© Rinton Press

AN APPROACH FOR WEB SERVICE DISCOVERABILITY ANTI-PATTERN DETECTION FOR JOURNAL OF WEB ENGINEERING

JUAN MANUEL RODRIGUEZ MARCO CRASSO ALEJANDRO ZUNINO
ISISTAN, Universidad Nacional del Centro de la Provincia de Buenos Aires, Paraje Arroyo Seco
Tandil, Buenos Aires B7001BBO, Argentina
CONICET
juanmanuel.rodriguez@isistan.unicen.edu.ar marco.crasso@isistan.unicen.edu.ar alejandro.zunino@isistan.unicen.edu.ar

Received February 17, 2012
Revised October 26, 2012

The Service Oriented Computing paradigm and its most popular implementation, namely Web Services, are at the crossing of distributed computing and loosely coupled systems. Web Services can be discovered and reused dynamically using non-proprietary mechanisms, but when Web Services are poorly described, they become difficult to be discovered, understood, and then reused. This paper presents novel algorithms and heuristics for automatically detecting common pitfalls that should be avoided when creating Web Services descriptions. To assess the accuracy of the proposed algorithms and heuristics, we compared their results with the results of manually analyzing a data-set of 400 publicly available services. In addition, we analyzed the correlation between the algorithms and heuristics results and other well-known quality metrics, which were presented by Al-Masri and Mahmoud. The average detection accuracy was 93.14% , and the false positive and false negative rates of 4.06% and 9.91% , respectively. Additionally, the Al-Masri and Mahmoud's quality metrics related to Web Services descriptions had a direct correlation with most of the automatic detecting results. The proposed algorithms and heuristics for automatically detecting common pitfalls are powerful tools for both improving existent Web Services and developing new Web Services that can be easily discovered, understood and reused.

Keywords: Web Services, Web Services Discoverability Anti-patterns, Web Services Modeling, Anti-patterns detection.
Communicated by: B. White & O. Díaz

1 Introduction

Creating software that utilizes information and services provided by third-parties is very common. Software companies tend to focus internal resources on developing core features, while delegating non-core operations to an external entity [1]. Nowadays, this kind of software usually is developed under a paradigm, called *Service-Oriented Computing* (SOC), in which developers look for independent loosely coupled third-party software pieces, called *services* [2].

When developing software under the SOC paradigm, there are three main roles: service consumer, service provider and service discovery system. The service provider offers services which are expected to be invoked by service consumers. To use a service, a service consumer needs to know that it is available. Helping service consumers to find the service that they need is the function of the service discovery system role [3].

Since services need to be interoperable, they are implemented using well-known Internet protocols [4]. These services are called *Web Services*. Web Services standard defines a registry interface and a language to describe the services, namely Universal Description, Discovery and Integration (UDDI)[a] and Web Services Description Language (WSDL),[b] respectively. Basically, an UDDI registry allows service providers to publish their services and service consumers to look for the services that they need. Publishing a service in UDDI consists on describing its provider, characterizing its functionality based on standard taxonomies and services' WSDL documents.

Several studies confirmed that Web Services are not as widespread (i.e., at global scale) as expected because finding the right service is a hard task. The discovery problem has been an issue of Web Services from their very beginnings because UDDI registry search capabilities are inappropriate for an extremely open and heterogeneous setting like the Internet [5]. Therefore, several search approaches, e.g. those based on the Semantic Web vision [6] or classic Information Retrieval (IR) techniques [7, 8], have been rapidly proposed to complement UDDI.

Except for the Semantic Web based discovery approaches, which utilize ontologies as additional specifications for the services, other discovery approaches only rely on information extracted from the UDDI registries entries, e.g. WSDL documents, associated with the services because them are supposed to contain information that specifies services' functionality. WSDL is an XML-based language for describing a service intended functionality using an interface with methods and arguments, in object-oriented terminology, and documentation as textual comments. Concretely, a WSDL document has words or terms, placed in the operation signatures and comments [9], that describe the service [8]. In consequence, several authors have adapted classic IR techniques [7, 9, 8], such as word sense disambiguation, stop-words removal, and stemming, for service discovery [8].

Although well-written WSDL documents are essential not only to developers, who need WSDL documents' information to invoke the service [10], but also to IR-based discovery systems, it seems that service developers tend not to care about the WSDL documents' quality. Several researchers [7, 11, 12, 13, 14, 15] have pointed out a wide range of recurrent WSDL document problems that negatively impact on both discoverability and usability of services. These efforts originate a research line for Web Services discoverability concerns.

As far as we known, one of the most exhaustive and complete survey on Web Services discoverability concerns is the work presented in [11]. The authors introduced a discoverability anti-patterns catalog that aims to help developers to improve their WSDL documents [11]. The catalog consists of eight anti-patterns and each anti-pattern has a name, a problem description, and a soundly refactor procedure. In addition, [11] analyzes how each anti-pattern affects the capability of not only service discovery systems to find a suitable service, but also humans to understand the service functionality [1]. Broadly, the results empirically confirm that the removal of discoverability anti-patterns makes services easier to be understood and discovered by potential consumers.

Despite having an anti-pattern catalog is useful, looking for anti-patterns in WSDL documents might be a time consuming and complex task [11]. Furthermore, developers tend to disregard WSDL document quality importance, even when they are developing enterprise systems [16]. Therefore, there is a clear need of automatic support to spot anti-patterns occurrences in WSDL documents, which allows service providers to improve their service descriptions prior to make them publicly available. Although some anti-patterns can be detected only by analyzing the structure, or syntax of the WSDL document, others require a more complex analysis, which can include analyzing the semantics of

[a] UDDI, `http://uddi.xml.org/`
[b] WSDL, `http://www.w3.org/TR/wsdl`

the WSDL document elements. Therefore, this paper presents algorithms to detect the simplest anti-patterns and heuristics for more complex anti-patterns. Our proposed heuristics combine techniques from the Machine Learning and Natural Language Processing areas, namely automatic document classification [17] and probabilistic context free grammar parsing [18, 19, 20], respectively.

Both the heuristics and algorithms have been implemented as a tool for developers. Given a WSDL document, the tool outputs a list of anti-pattern occurrences if any could be detected. The detection effectiveness of the algorithms has been experimentally validated with a 392-WSDL document data-set [21], which is publicly available. At the same time, the heuristics effectiveness has been tested using the same data-set, but also the achieved results have been correlated with those showed in [22], a survey of quality metrics from 2250 publicly available Web Services that includes metrics concerned with WSDL documents discoverability.

The main contributions of this paper are:

- the definition of algorithms and heuristics for detecting frequent practices that hinder Web Services discoverability,
- experiments showing that the proposed algorithms and heuristics enable the detection of anti-patterns within WSDL documents.
- assessments about the correlation between these anti-patterns and well-known Web Services quality metrics.

The rest of this paper is organized as follows. In Section 2, we describe how Web Services are specified using WSDL documents, and methods for automatically assessing the quality of a Web service in terms of discoverability. Next, in Section 3, the anti-pattern catalog, on which this work relies, is briefly described. Section 4 presents the proposed approach to detect the anti-patterns in a WSDL document. Then, Section 5 assesses the effectiveness of the proposed approach. Future research possibilities are presented in Section 6. Finally, Section 7 concludes the paper.

2 Background

On one hand, our approach works over service descriptions, thereby it is needed to understand how services are described. At the same time, since our work aims at helping services owners to improve service description quality, and service description are software artifacts, it is necessary to appreciate why traditional software quality metrics are not applicable in this context. Therefore, this section introduces two main topics: how Web Services are described and traditional software quality metrics.

2.1 Describing Web Services

WSDL is a language that allows developers to describe two main parts of a service: its functionality, and how to invoke it. Conceptually, a WSDL document reveals the service interface that is offered to the outer world. The latter part specifies technological aspects, such as transport protocol and network address. Discoverers use the functional descriptions to match third-party services against their needs and, in turn, they take under consideration the technological details for invoking the selected service.

Technically, a WSDL document describes the service functionality as a set of *port-types*, which arrange different *operations* whose invocation is based on *message* exchange. Messages stand for the inputs or outputs of the operations, indistinctly. Exceptions are described as ordinary messages called *faults*. Besides, the main elements of a WSDL document, such as port-types, operations and

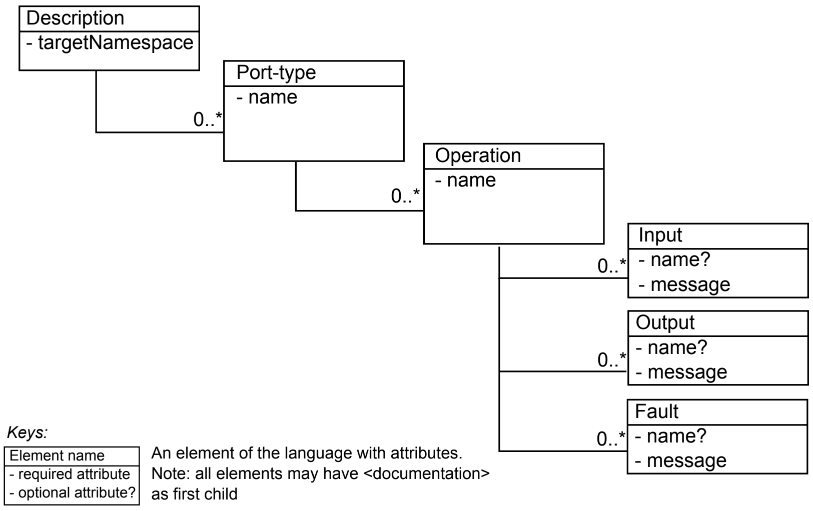


Fig. 1. WSDL v1.1 Infoset.

their messages, must be named with unique identifiers. Figure 1 depicts the Information set (Infoset) diagram of the WSDL for service interfaces. We used the WSDL specification version 1.1 through this paper because it is the most currently used, yet WSDL specification version 2.0 is not different from version 1.1 in terms of the information that a WSDL document contains about a service interface.

Messages consist of *parts* that transport data between services' consumers and providers, and vice-versa. Each message part is arranged according to specific data-type definitions. The XML Schema Definition (XSD)[c] language is employed to express the structure of message parts. XSD offers constructors for defining simple types (e.g., integer and string), restrictions and both encapsulation and extension mechanisms to define more complex elements.

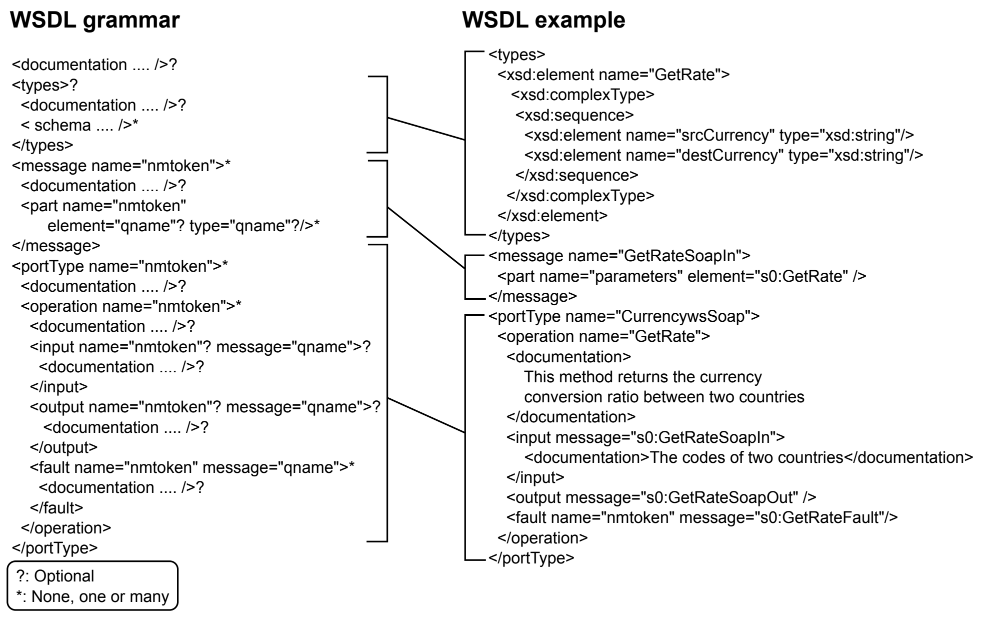
For the sake of exemplification, the right side of Figure 2 shows the WSDL grammar, while a real world example is shown at the left side of the figure. Chosen example contains a port-type with only one operation. The operation is named GetRate and it is designed for returning the currency conversion rate between two countries, as it is described by its associated documentation. The GetRate operation expects a message containing two country codes as input. As the reader can see, the types tag includes the code needed for representing a complex data-type, called "CountryCodes" which is exchanged in the input message of the "GetRate" operation. Alternatively, XSD code might be put into a separate file and imported from the WSDL document or even other WSDL documents afterward.

2.2 Measuring software code quality

Researchers have study software code quality from the very beginnings of software engineering [23, 24]. Usually, the software code quality is expressed through metrics that analyze different aspects of the same source code artifact. For instance, there are metrics for component coupling, documentation, or complexity. In addition to defining these metrics, researchers also have proven that these metrics can be used to predict the amount of failures in a software component [25, 26].

These metrics can be as simple as counting the lines of code. However, most of these metrics [27, 25, 26] requires not only to process a module header, but also its associated code in order to measure

[c] XML Schema Part 0: Primer Second Edition, `http://www.w3.org/TR/xmlschema-0/`



```
<documentation .... />?
<types>?
  <documentation .... />?
  < schema .... />*
</types>
<message name="nmtoken">*
  <documentation .... />?
  <part name="nmtoken"
        element="qname"? type="qname"?/>*
</message>
<portType name="nmtoken">*
  <documentation .... />?
  <operation name="nmtoken">*
    <documentation .... />?
    <input name="nmtoken"? message="qname">?
      <documentation .... />?
    </input>
    <output name="nmtoken"? message="qname">?
      <documentation .... />?
    </output>
    <fault name="nmtoken" message="qname">*
      <documentation .... />?
    </fault>
  </operation>
</portType>
```

```
?: Optional
*: None, one or many
```

```
<types>
  <xsd:element name="GetRate">
    <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="srcCurrency" type="xsd:string"/>
       <xsd:element name="destCurrency" type="xsd:string"/>
     </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</types>
<message name="GetRateSoapIn">
  <part name="parameters" element="s0:GetRate" />
</message>
<portType name="CurrencywsSoap">
  <operation name="GetRate">
    <documentation>
       This method returns the currency
       conversion ratio between two countries
    </documentation>
    <input message="s0:GetRateSoapIn">
       <documentation>The codes of two countries</documentation>
    </input>
    <output message="s0:GetRateSoapOut" />
    <fault name="nmtoken" message="s0:GetRateFault"/>
  </operation>
</portType>
```

Fig. 2. Web Services Description Language grammar and example.

them. Here, we refer to module as a general term for denoting functions, procedures, methods, and any block of code exposing a signature. For instance, cyclomatic complexity [23] requires creating a graph that represents all the possible executions flows of a software component. Although these metrics can be automatically calculated, they are strongly correlated with more abstract concepts [28], such as cohesion, which is the semantic relation between components of the same software artifacts. Finally, this kind of metrics need the source code to perform a calculation. In our study, since we only have the WSDL documents of services, these metrics are not applicable.

Other researchers have attempted to measure directly more abstract concept, like cohesion or documentation quality. In [29], they present a tool, called JavadocMiner, that implements set of heuristics and algorithms to analyze Javadocs, which are how Java classes interfaces are documented. This work bases on the fact that by using Javadoc, methods comments are structured in a known way and explicitly references to methods inputs, return and exceptions are included. In consequence, JavadocMiner uses this structure to verify whether all the elements are referenced in the comments or not. In addition, JavadocMiner employs other simple heuristics, such as tokenized nouns and verbs, counting words per Javadoc, and calculating readability formulas [30], for evaluating the quality of the present documentation.

Researchers have also applied linguistic metrics, natural language processing techniques, and heuristic rules to study the quality of identifiers and comments in source code. In [31], the authors propose to use well-known text metrics to detect anti-pattern in comments, while other authors [32] only employ simple rules to detect problems with identifiers. In [33] the authors present a detailed understanding of Java class identifier naming conventions, showing most relevant naming patterns and grammatical structures; in [34] the authors show an approach to automatically improve the descriptiveness of labels used to name business entities.

Finally, several authors have proposed measuring cohesion using graphs that represent how different elements of the code artifacts are related [27, 35, 36]. An example of this is the Lack of Cohesion on Method [27] that is the number of pair methods which share no instance variables. These methods for measuring cohesion are based on source code, yet they are not useful when the only available information is the system interface, which is very common when using Web Services.

To sum up, traditional software quality metrics are, at best, difficult to be adapted for measuring WSDL document quality. Due to this difficulty, it is necessary to develop new heuristics for automatically assessing the discoverability of Web Services. However, we have adapted or reused, when it was possible, some of the bases of these metrics in our approach to automatically detect anti-patterns.

3 Related work

Researchers have studied Web Services quality from different angles, such as interoperability, security and discoverability, or interface quality. Regarding both interoperability and security, the Web Services Interoperability Organization (WS-I), an organization that establishes best practices for interoperability, has released several profiles.[d] A profile describes how to implement interoperable services based on the exhaustive analysis of popular enterprise software middlewares, namely .NET and JEE. In addition, the WS-I provides testing tools, e.g., BP 1.2 and 2.0 Interoperability Test Suites, to ensure that a Web Service implementation satisfies these profiles. Other research works aim to improve Web Services performance [37] and reliability [38]. Finally, several efforts aim to analyze the quality of WSDL documents [11, 13, 14]. These works are further described below because they are very near to the approach presented in this paper.

In [14], the authors analyzed the comments present in a real-life WSDL document data-set. To analyze those WSDL documents comments, the authors crawl public registries on the Internet. Initially, the authors gathered 2432 Web Services, but this number decreased to 1544 because some registered services have a non-valid URL to their WSDL documents. Moreover, the authors discarded duplicated services and the ones with an invalid WSDL document, i.e., a bad-formed WSDL document. Thus, the number of gathered services fell to 640. Then, the authors analyzed the lengths of the textual comments of these services (including the registration information and all the comments present in the WSDL documents). The paper shows that in the 80% of 640 real-life WSDL documents the average documentation length for operations is 10 words or less. Furthermore, from this 80%, half of them have no documentation at all. In addition, 80% of the comments have less than 50 words, while 52% of the comments have less than 20 words. Afterward, the authors averaged the lengths of the textual comments associated with <operation> elements in the WSDL documents.

In [13], the authors analyze elements names in WSDL documents. The authors detect “naming tendencies” in the names of WSDL document elements and empirically show that these tendencies negatively impact the retrieval effectiveness of a syntactic registry. The experiment consisted in analyzing the names of message parts that belong to 596 WSDL documents gathered from Internet repositories. The name of each message part was compared against the rest of part names, then four naming tendencies were observed in service message parts. Broadly, these tendencies show that developers use common phrases within part names. For example, a message part standing for a user’s name is commonly called “name”, “lname”, “user_name” or “first_name” [13]. Finally, the paper empirically shows that the retrieval effectiveness of a syntactic registry can be improved by enhancing its underlying matching approach for dealing with the observed tendencies.

The work presented in [11], in which this paper is heavily based, explicitly pursues recurrent

[d] Basic Profiles: `http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile`

Table 1. Web Services discoverability anti-patterns.

Anti-pattern	Occurs when	Manifest
Whatever types	A special data-type is used for representing any object of the problem domain.	Evident
Redundant port-types	Different port-types offer the same set of operations.	Evident
Redundant data models	Many data-types for representing the same objects of the problem domain coexist in a WSDL document.	Evident
Enclosed data model	The data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD ones.	Evident
Inappropriate or lacking comments	(1) a WSDL document has no comments, or (2) comments are inappropriate and not explanatory.	(1) Evident, or (2) Not immediately apparent
Low cohesive operations in the same port-type	Port-types have weak semantic cohesion.	Not immediately apparent
Ambiguous names	Ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent
Undercover fault information within standard messages	Output messages are used to notify service errors.	(1) Present in service implementation, or (2) Not immediately apparent

problems that attempt against the understandability and discoverability of a service. To do this, the authors analyzed 392 WSDL documents, which had been gathered from the Internet, and found that the functionality of many of them would be hard to be understood and that some patterns of WSDL code were present in them. As a result, the study presents a catalog of bad practices that frequently occur in this public corpus of WSDL documents. This catalog not only supplies each problem with a practical solution, but also provides hints about how to detect problem symptoms, thereby it is an *anti-patterns* catalog.

Table 1 summarizes anti-patterns presenting their names, when they occur and how they manifest themselves. An anti-pattern manifestation can be: *Evident*, *Not immediately apparent*, and *Present in service implementation*. An anti-pattern is *Evident* if it can be detected by only analyzing the structure or syntax of a WSDL document. *Not immediately apparent* means that detecting the anti-pattern requires not only a syntactical analysis but also a semantic analysis of the textual information. Finally, *Present in service implementation* anti-patterns might not show themselves in the WSDL document, thus requiring the execution of the associated service to be detected. Anti-pattern manifestation is highly important for this paper because the way an anti-pattern manifests itself drives the approach to detect the anti-pattern, as will be explained in next Section.

4 An approach to automatically detect WSDL anti-patterns

This section describes an approach to automatically detect the eight WSDL anti-patterns summarized in Table 1. Roughly, this approach consists of one independent detection algorithm or heuristic, a.k.a. *detectors*, per anti-pattern. Since anti-pattern occurrences are independent from the other anti-patterns occurrences [11], each anti-pattern has its independent detector. Therefore, analyzing a WSDL document is an incremental process that executes eight different detectors, i.e., one per anti-pattern.

The anti-patterns detectors can be classified using the anti-pattern manifestation taxonomy shown in the third column of Table 1, because how an anti-pattern manifests itself is strongly related to the analysis required for detecting it. Evident anti-patterns are structural anti-patterns related to redundant WSDL elements in a WSDL document or WSDL elements with a particular structure, such as a type that allows any valid XML (*Whatever types* anti-pattern). These anti-patterns can be detected simply by applying detection rules to the WSDL document structure.

In contrast, Not immediately apparent anti-patterns affect the semantics within the comments and names in the WSDL document rather than its structure. Basically, Not immediately apparent anti-patterns detection requires an analysis of the meaning of names and comments in a WSDL document. As a consequence, our heuristics to detect these anti-patterns are based on several well-known natural language processing techniques [18, 19, 20, 39, 40].

The *Undercover fault information within standard messages* anti-pattern is classified as both Not immediately apparent and Present in service implementation meaning that this anti-pattern might have no footprint in a WSDL document. However, since we aim to detect anti-patterns using WSDL documents, the presented heuristic for detecting this anti-pattern is only applicable when the anti-pattern manifests itself in the WSDL document –this is when the anti-pattern is classified as Not immediately apparent–. Although this can be seen as a limitation of our approach, the lack of footprint in a WSDL document is usually a result of another anti-pattern symptom. Hence, solving other anti-patterns might unveil *Undercover fault information within standard messages* anti-pattern footprint, which is consistent with the idea of analyzing the WSDL documents in an incremental way. For example, *Ambiguous names* anti-pattern can hide *Undercover fault information within standard messages* anti-pattern because names related to output messages are the only footprint of this anti-pattern in WSDL documents. This means that trying to detect the *Undercover fault information within standard messages* anti-pattern before removing the *Ambiguous names* anti-pattern might not be effective, but if the WSDL document names are representative, the detection technique for *Undercover fault information within standard messages* anti-pattern would be more effective. In the following sections, the techniques for detecting anti-patterns are discussed.

4.1 Algorithms for detecting Evident anti-patterns

In this section, we describe the algorithms for detecting Evident anti-patterns in a WSDL document. Evident anti-patterns are: *Lacking comments, Enclosed data model, Redundant port-types, Redundant data models and Whatever types*. Their associated detectors rely on rules based on the WSDL grammar, because Evident anti-patterns manifest themselves in the WSDL document structure.

Since service developers often do not include comments in WSDL documents [14], the first detector that we present is *Lacking comments* detector. Detecting the lack of comments consists in checking that all operations have associated the <documentation> tag, and if it is present, checking whether its content is not an empty string, which means that there is some documentation.

The second detector is the *Enclosed data model* detector. To detect the *Enclosed data model* anti-pattern, it is necessary to know whether the data model is defined within the WSDL document or imported from somewhere else. Basically, the algorithm firstly checks whether the <types> tag is present in a WSDL document. If it is not present, the data model is not defined in the WSDL document; thereby the anti-pattern is not present. Although when the WSDL document contains the

<type> tag, there are two cases: the <types> tag is empty, or contains one or more <schema> tags. If the <types> tag is empty, it again means that the WSDL document does not define data types. Finally, if the <types> tag has <schema> tags defined, it is necessary to check each <schema> tag. If all <schema> tags are empty, the anti-pattern is not present, otherwise it is present.

The *Redundant port-type* detector checks that all port-types in the WSDL document are unique. The detector compares every port-type in the WSDL document against the others. To compare two port-types, our algorithm checks whether both port-types have the same number of operations and if they have the same name in both port-types. Since the same operations in different port-types tend not to share messages, our algorithm skips message similarity checks. This is because, when this anti-pattern is present, the developers tend to define different messages for the different transport protocols. Therefore, the input message for an operation of the SOAP port-type is not the same as the input message for the same operation in the HTTP-GET port-type.

Another recurrent problem that also involves repeated code is the *Redundant data model* anti-pattern. In this case, the detector compares the structure of all the data types to verify that they are unique. In this comparison, the data type names are ignored because they tend to be different. Therefore, only the schema structure is taken under consideration.

Finally, the *Whatever type* anti-pattern means that one or more of the defined types allow exchanging a generics type [41]. The *Whatever type* detector verifies that there is no primitive type “anyType” or <any> tag within the schema. The existence of any of these ways for defining generic types means that the anti-pattern is present.

4.2 Heuristics for detecting not immediately apparent anti-patterns

Other anti-patterns do not manifest themselves in a WSDL document structure, but in the semantics of its comments and names, thus these anti-patterns are categorized as Not immediately apparent [11]. The anti-patterns in this category are *Low cohesive operations in the same port-type*, *Ambiguous names*, *Inappropriate comments* and *Undercover fault information within standard messages*. Below, the detector associated with each anti-pattern is explained.

4.2.1 Low cohesive operations in the same port-type detector

Sometimes, two or more operations that are not semantically related are in the same port-type. This situation renders the *Low cohesive operations in the same port-type* anti-pattern. Therefore, our heuristic automatically rules out any port-type with only one operation. However, if a port-type has two or more operations, the detector needs to compare each another to verify that they are semantically related. As the source code implementing third-party Web Services is not available, one approach to automatically determine whether two or more operations are related consists in classifying them according to keywords present only in their names and associated comments. Then, operations are considered as being related when they belong to the same class or domain.

The information available about the operations is mostly their names, comments, message and data type names; all this is textual information. As a consequence, the problem of classifying the operations according to their domain can be mapped to the problem of classifying text according to its domain. Actually, several techniques can be used to classify text [42]. We selected a variation of the Rocchio classifier [40], called Rocchio TF-IDF. We choose this classifier because a previous research [17] has empirically proved that Rocchio TF-IDF have a better performance than other classifiers when classifying WSDL documents.

The first step to classify an operation is to convert it to a vector representation in which each dimension stands for a term and its magnitude is the importance of the terms in the operation. Firstly, the terms extracted from the operations are pre-processed to remove irrelevant terms [43], known as stop-words, and obtain the stems of the word instead of using them as they are in the WSDL document. Pre-processing the terms helps to reduce the dimension of the problem without introducing significant information lost, and in turn boosts the accuracy of the Rocchio classifier. In particular, our detector uses the English version of the Porter stemming algorithm [44]. Finally, the TF-IDF technique is applied to calculate the weight of each dimension because several works [9, 8] have proved the TF-IDF technique effectiveness in for Web Services descriptions [9, 8].

The term TF-IDF stands for term frequency–inverse document frequency, meaning that the weight of a term depends not only on the number of times that the term is in a document, but also on the number of documents of the entire corpus that have the term. Formally: $tfidf = tf \bullet idf$ with:

$$tf_i = \frac{n_i}{\sum_k n_k}$$

with n_i being the number of occurrences of the considered term, the denominator is the number of occurrences of all terms, and:

$$idf_i = log \frac{|D|}{|d : dt_i|}$$

where $|D|$ is the total number of documents in the corpus and $|d : dt_i|$ is the number of documents where the term (t_i) appears.

In the Rocchio classifier, categories are represented as vectors as well. The vector for a category $(\vec{\mu}(c))$ is calculated as the center of mass of already known vectors of documents in that category (D_c). In this case, the documents are WSDL documents that experts have classified. Each vector $\vec{v}(d)$ is calculated by extracting terms not only from the word of one operation, but also from the whole WSDL document. Formally, a category center of mass is calculated as follows:

$$\vec{\mu}(c) = \frac{1}{D_c} \sum_{d \varepsilon D_c} \vec{v}(d)$$

Having the center of mass of the categories, an operation is considered to belong to the category that its center of mass is the closest to the operation vector. This similitude is measured using the cosine similarity. Given two vectors (A, B), their similitude is:

$$cos(A, B) = \frac{\sum A_i \cdot B_i}{\sqrt{\sum A_i^2} \cdot \sqrt{\sum B_i^2}}$$

In this way, we use Rocchio TF-IDF algorithm to classify operations into pre-defined classes, which represents the operations domains. Our detector verifies whether a port-type contains operations that belong to different domains, which means that the anti-pattern is present. The main disadvantages of using Rocchio TF-IDF are that the classifier is only able to classify operation in known domains, which requires an expert to classify a train set of operations or WSDL do, and depends on the language. Algorithm 1 shows how a WSDL document is analyzed to determine whether it has a low cohesive operation in any of its port-types.

Algorithm 1 Low cohesive operations in the same port-type anti-pattern detection heuristic.

```
function hasLowCohesivePortTypes($wsdlDocument, dataset$)          ▷ Returns a Boolean
    $classifier \leftarrow$ TrainRocchio($dataset$)
    $portTypes \leftarrow$ getPortTypes($wsdlDocument$)
    for all $portType \in portTypes$ do
        $operations \leftarrow$ getOperations($portType$)
        $size \leftarrow$ Size($operations$)
        if $size > 1$ then
            $operation \leftarrow operations$.removeFirst( )
            $\vec{o} \leftarrow$ CreateVector($operation$)
            $class \leftarrow classifier$.Classify($\vec{o}$)
            for all $operation \in operations$ do
                $\vec{o} \leftarrow$ CreateVector($operation$)
                $newClass \leftarrow classifier$.Classify($\vec{o}$)
                if $newClass \neq class$ then
                    return true
                end if
            end for
        end if
    end for
    return false
end function
```

4.2.2 Ambiguous names detector

Element names in a WSDL document are important because names not only provide relevant terms for IR-based registries, but also are essential for service consumers, who want to understand the services. Although, it is usual that developers select names that are not the best [11, 13]. Problems in naming are described by the *Ambiguous names* anti-pattern.

The first characteristic to analyze in the names is their length because one of the main issues with names in WSDL documents is that they tend to be too short or too long. If the length of the name is between 3 and 30 characters, the name is considered to have an adequate length [13]. Otherwise, the name is considered as bad name that should be changed.

The second issue detected is that many names use several words that are non-explanatory or too general [11, 13]. Many of those words are repeated across several services and they are among the most common WSDL document element names [11]. The unrecommended words are: *thing, class, param, arg, obj, some, execute, return, body, foo, http, soap, result, input, output, in* and *out.* A name that has any of these words probably is too general, meaning that the name is probably an ambiguous name.

Finally, the grammatical structure of the names must be accord to their purpose. Names for operations, which are actions applied over the input parameters, should start with a verb that describes the operation action. Furthermore, an operation name should only have one verb, because an operation should perform only one action. In contrast, message part names should be nouns or noun phrases, because message parts are the things over which the operations are applied.

Then, our detector performs this name grammatical analysis on the operations' names and message parts' names. This analysis is made using a probabilistic context free grammar parser [18, 19, 20] based on parsing rules as traditional context free grammar, but each rule has associated a probability of occurrences, which is independent form the probability of other rules. Therefore, a single sentence might have associated several parsing trees and each tree has an associated probability, which is the product of all parsing rules' probability. Having the probability of each tree, our heuristic selects the most likely tree to perform the analysis.

The analysis of the operations' and message parts' names is slightly different. When analyzing an operation name, the heuristic adds the word "it", which indicates the noun that should be missing in the operation name, at the name beginning. For instance, if the operation is named "buyCar", the sentence analyzed by the parser is "it buy car". Although the sentence is not grammatically correct, it is close enough to a correct sentence and the parser will be able to handle it. This is due to the fact of having probabilistic rules, which allows the parser to handle malformed sentences [18].

Operations' names are expected to have one and only one verb placed at the beginning and after it a noun or noun phrase, thus if the number of verbs is different from one, the *Ambiguous name* anti-pattern is detected. Figure 3 represents parsing trees for three different operation names with the "it" pronoun added as explained above. The first name, which is "buyCar", is correct because it gives the idea that the operation performs only one action. In contrast, the second name, which is "car", is a noun that is incorrect because the name has not semantics of what the operation does. Finally, the third operation name, which is "createSendTicket", is also incorrect because, despite beginning with a verb, it has another verb giving the impression that the operation performs two actions.

In the second case, the name of a message part is provided to the parser unmodified because the parser has to determine whether it is a noun or a noun phrase. A message part name parsing tree should not contain any verb tag because a verb indicates that the message part modifies the operation behavior. Figure 4 depicts the parsing tree of three message part names. The first and second names, which are "name" and "firstName" respectively, have no problem because they represent a thing. However, the third name, which is "usesCache", starts with a verb and it is probable indicating if the operation should be executed with the information in the cache or not. This is contrary to the well-known black-box operation good practice [24] because the user must know how the operation is implemented in order to invoke it.

4.2.3 Inappropriate comments detector

In addition to names, comments are an important WSDL document part that provides semantic information because they not only supplies Web Services registries with terms, but also helps a potential service user to understand the service functionality. However, badly commented WSDL documents

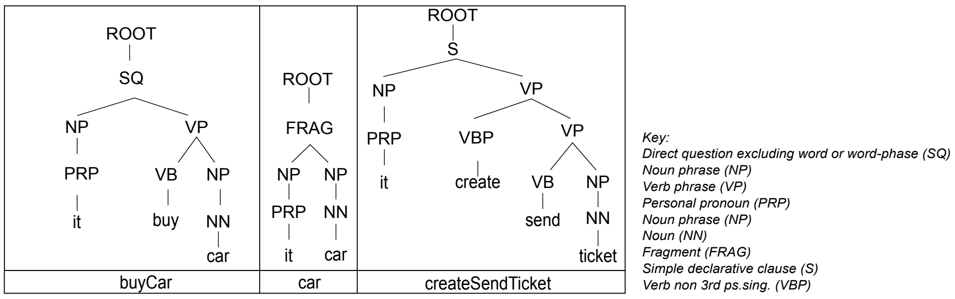


Fig. 3. Parsing tree of operation names.

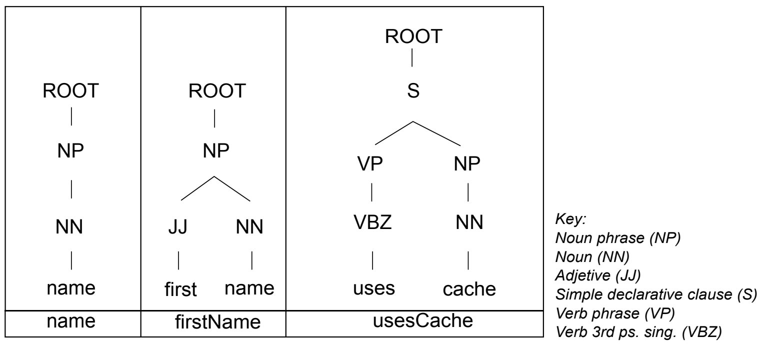


Fig. 4. Parsing tree of message part names.

are extremely common [14]. Detecting whether a comment is inappropriate requires understanding the comment semantics as well as the operation semantics. To analyze a comment semantics, it is necessary to have a language corpus [39] as well as some sense disambiguation algorithm [45, 46].

To detect inappropriate comments, our heuristic assumes that a WSDL document has all its operations commented. For analyzing the comments, our heuristic relies on a corpus that describes English words and their relationships known as WordNet [39]. Using this corpus, we propose a simplified version of two sense disambiguation algorithms [45, 46]. In particular, our algorithm uses the WordNet hypernym relationship to disambiguate word meaning. A word hypernym is another word or phrase with a more wide-ranged concept that includes the concept represented by the first one. For instance, in WordNet a “car” hypernym is “motor vehicle, automotive vehicle” which also includes, according to WordNet, the concepts related to “motorcycle, bike”, “truck, motortrack” or a “golfcart”. The hypernym relationship can be seen as a tree of word and phrases in which the root node is the most general concept, e.g., “entity” in the cases of a noun. In these trees, leaves are very specific concepts, while the root is a generic one.

The sense disambiguation generation tree algorithm is depicted in Algorithm 2; it is based on combining the trees of different words using a occurrence counter to keep track of how frequent a concept is. The algorithm creates the resulting tree in an incremental manner, at line 2 starts with a tree that has only one node (all) that is the common root for all the word tree, and combines that tree to the tree associated with each term one by one. The tree goal is to have weighted all the concepts that can be related to the documentation. Since the weight of a node is the sum of its children weight or the sum of the word occurrences if it is a leaf, the nearer the node is to the root, the higher its weight is.

The detection heuristic relies on these trees to identify the semantics of the documentation. Basically, the heuristic generates two trees for the comments: one for the nouns and other for the verbs and uses them with two purposes. The first purpose is to ensure that the comments are not too general. The second one is to compare the documentation semantics with the operation semantics.

To measure the comments specificity, our heuristic verifies whether the noun and verb trees depth for the comments are higher than an experimentally selected threshold. The minimum noun-tree deep threshold (NDT) is 5, while the minimum verb-tree deep threshold (VDT) is 1. The values of NDT and VDT were empirically determined using the methods that will be described in Section 5.1.1. Our heuristic measures the similitude between the noun and verb trees associated to the operation name and comments. The similitude between two trees is defined as the depth of the longest-most-

Algorithm 2 Combining tree.

```
1:  function createTree($wordList$)
2:      $treeRoot \leftarrow$ createNode(ALL)
3:      for $word$ in $wordList$ do
4:          $wordTree \leftarrow$ WordNetGetHypernymTree(word)
5:          combineTree($treeRoot, wordTree$)
6:      end for
7:      return treeRoot
8:  end function
9:  procedure combineTree($result, newTree$)
10:     for $child$ in $newTree.children$ do
11:         if isChildOf($child, result$) then
12:             getEqualNode(result,child).$counter++$
13:         else
14:             $newChild \leftarrow$ createNode(child)
15:             $newChild.counter = 1$
16:             addNewNode($result, newChild$)
17:         end if
18:         $nextResult \leftarrow$ getEqualNode(result,child)
19:         combineTree($nextResult, child$)
20:     end for
21: end procedure
```

frequent path shared by both trees divided by the depth of the longest tree. Algorithm 3 depicts how the longest-most-frequent path shared by two trees is recursively calculated. Finally, this similitude degree is compared to an experimentally selected threshold. The selected value for the noun similarity threshold (NST) and verb similarity threshold (VST) is 10% in both cases. The reasons for selecting these thresholds will be described in Section 5.1.1.

4.2.4 Undercover fault information within standard messages detector

Finally, the *Undercover fault information within standard messages* anti-pattern belongs not only to this category, but also to the Present in service implementation one. But since the input for the heuristic is a WSDL document, the heuristic is designed to detect the anti-pattern only if it has a footprint in the WSDL document, which is most likely when the WSDL document is not affected by other anti-patterns. Firstly, the detector verifies whether the operation has a <fault> message defined that means that the errors are handled in the correct manner or not. Consequently, the presence of a <fault> message is considered enough evidence that the operation presents no symptom of the anti-pattern. If this not the case, the heuristic looks for keywords that indicate the presence of the anti-pattern in the operation documentation, output message name and names of the data types referred by the output message. The set of keywords is: *error, errors, fault, faults, fail, fails, exception, exceptions, overflow, mistake and misplay*. If the detector founds some of these words, the WSDL document is considered to be affected by the *Undercover fault information within standard messages* anti-pattern. Although this anti-pattern is related to the output semantics, it can be detected following these simples rules because the keywords are practically a convention in most of the programming languages/platforms, such as Java, .Net or C++, used to provide an implementation of the target services.

Algorithm 3 Longest-most-frequent path shared.

```
function LONGESTMOSTFREQUENTPATH($tree1, tree1$)
    $result \leftarrow 0$
    for $child1$ in $tree1.children$ do
        if HASTHEHIGHESTCOUNTER($child1, tree1.children$) then
            $child2 \leftarrow$ GETEQUALNODE(tree2,child2)
            if HASTHEHIGHESTCOUNTER($child2, tree2.children$) then
                $newResult \leftarrow$ LONGESTMOSTFREQUENTPATH(child1,child2)
                if $newResult > result$ then
                    $result \leftarrow newResult$
                end if
            end if
        end if
    end for
    RETURN($result$)
end function
```

5 Experimental evaluation

In the previous section, we have presented algorithms and heuristics for automatically detecting the discoverability anti-patterns described in [11]. This section describes three experiments that have been conducted to evaluate the proposed detection algorithms and heuristics' effectiveness of this proposal. The first experiment consisted of analyzing a data-set of real world WSDL documents using manual and automatic approaches and, in turn, comparing achieved results to assess the precision of the automatic approach.

Then, the second and third experiments consisted of studying the statistical relationship between the results of using the *Inappropriate or lacking comments* anti-pattern detector and the quality metrics presented in [22]. This is because this anti-pattern not only is one of the most difficult to detect and the most important for service discovery [1], but also is highly relevant for both service registries and service consumers [11]. In this sense, the second experiment presents a correlation analysis, whereas the third experiment further evaluates the effectiveness of the detection heuristic.

5.1 First Experiment: Manual vs. Automatic detection

In order to determine the effectiveness of our detectors, we have employed a 392 WSDL document data-set, which is available upon request. The WSDL documents in the data-set were collected by Hess et al. [21] from public Internet repositories, thereby they represent how Web Services are implemented in real life. For the manual detection, expert service developers analyzed the data-set to identify the anti-patterns in each WSDL document by hand. These results were peer-reviewed to assure their quality. At least three different people reviewed each WSDL document. This analysis is a revised version of the results presented in [11]. Figure 5 depicts the anti-pattern occurrences frequency according to our manual analysis.

Once we had the results of manually analyzing the data-set, we applied the automatic detectors on it, and finally compared both manual and automatic results. These results were organized per anti-pattern, in which if a WSDL document has the anti-pattern it is classified as "Positive", otherwise it is classified as "Negative". When the manual classification for a WSDL document is equal to the automatic one, it means that the detector has accurately operated for that WSDL document.

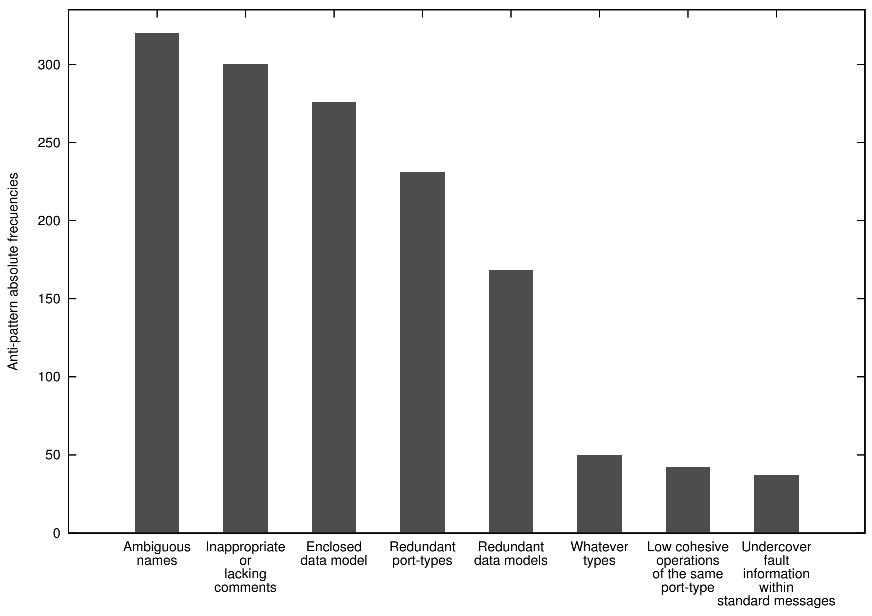


Fig. 5. Anti-pattern frequency.

The achieved results are shown using a confusion matrix per anti-pattern. The rows of the matrix represent the results of the automatic detector, while the columns of the matrix show manual classifications results. Using these confusion matrices, we assessed the accuracy, and false positive/negative rates for each matrix. Table 2 shows the confusion matrices.

The accuracy of each detector was calculated as the number of classification matchings over the total of analyzed WSDL documents. For instance, the accuracy of the detector associated with the *Enclosed data model* anti-pattern was $\frac{116+270}{116+5+1+270} = 98.47\%$. The heuristic for detecting *Low cohesive operations within the same port-type* anti-pattern achieved the lowest accuracy: 77.55%. Nevertheless, the averaged accuracy was 93.14%.

The false positive rate is the proportion of WSDL documents that a detector wrongly labels as having an anti-pattern. At the same time, the false negative rate is the proportion of WSDL documents that a detector wrongly labels as not having an anti-pattern. A false negative rate equal to 1 would mean that a detector missed all anti-pattern occurrences. For these rates, the lower the achieved values the better the detection effectiveness. The averaged false positive rate was 4.064%, and the averaged false negative rate was 9.911%.

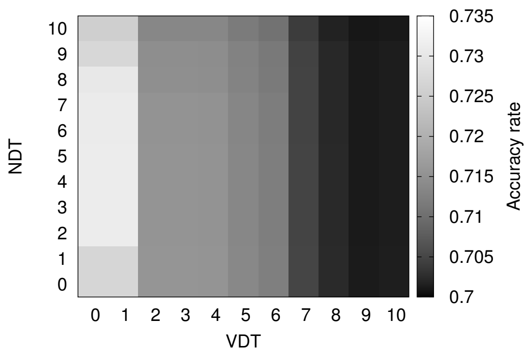
All in all, it can be seen that the accuracy values achieved by the algorithms-based detectors were higher than those achieved by heuristics-based detectors, which was expected. However, the accuracy value for these later was above 77.55%.

5.1.1 Inappropriate or lacking comment detector threshold determination

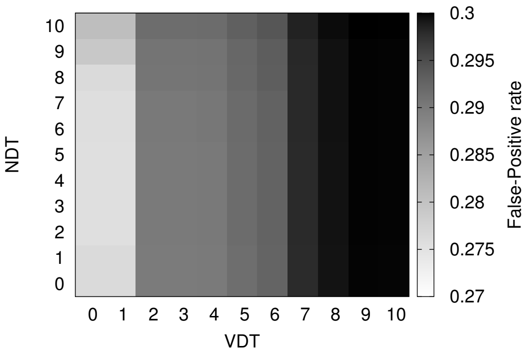
The Inappropriate or lacking comment anti-pattern detector has four parameters that have to be determined before running it. As explained in Section 4.2.3, these parameters are the thresholds for NDT, VDT, NST and VST. We followed an exploratory approach to select these thresholds, this is, we executed the *Inappropriate or lacking comments* anti-pattern detector using different thresholds

Table 2. Confusion matrix for the detection of anti-patterns.

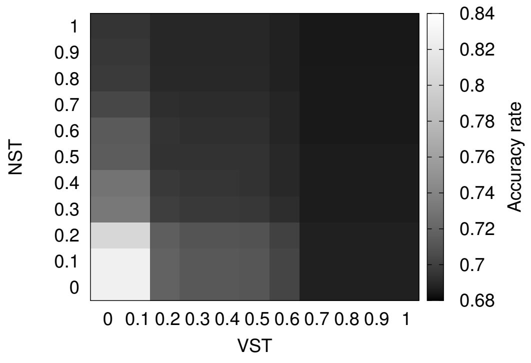
Automatic detection results per anti-pattern		Manual detection results		Statistical indicators		
		Negative	Positive	Accuracy	False Negative	False Positive
Enclosed data model	Negative	116	6	98.47%	2.17%	0%
	Positive	0	270			
Redundant port-types	Negative	161	4	98.97%	1.73%	0%
	Positive	0	227			
Redundant data model	Negative	221	2	98.72%	1.19%	1.34%
	Positive	3	166			
Whatever types	Negative	339	0	99.23%	0%	0.09%
	Positive	3	50			
Inappropriate or lacking comment	Negative	107	20	91.07%	12.29%	7.40%
	Positive	15	250			
Low cohesive operations in the same port-type	Negative	272	10	77.55%	23.80%	22.29%
	Positive	78	32			
Undercover fault information within standard messages	Negative	351	3	98.21%	8.11%	1.13%
	Positive	4	34			
Ambiguous names	Negative	7	3	82.90%	30%	0.26%
	Positive	1	381			



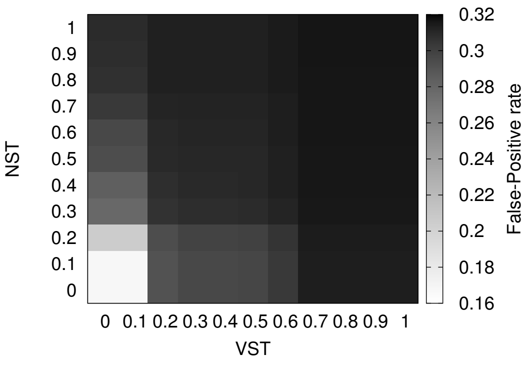
(a) {NDT,VDT} accuracy



(b) {NDT,VDT} false positive



(c) {NST,VST} accuracy



(d) {NST,VST} false positive

Fig. 6. Heat maps.

settings on the data-set described above. For each combination of parameters, we ran the detector, calculated its results, and compared it with the manually achieved results. Finally, the combination whose results were nearest to the manually obtained ones was used for the experiment described in previous section. For the sake of experimental reproducibility, the thresholds that we finally selected were: NDT=5,VDT=1, NST=0.1 and VST=0.1.

For both the NDT and VDT thresholds, we tested values ranging from 0 to 10, including the extreme values, with steps of 1. In contrast, for NST and VST we tested values from 0 to 1, with steps of 0.1. Therefore, we tried 14641 (i.e. 1121^4) different threshold settings and determined which combination worked better.

In order to better understand the results, we reduced the dimension of the problem by performing two analyses. First, we separated the 4 variables into 2 sets of 2 variables, namely {NDT, VDT} and {NST, VST}. Since for each possible instance of {NDT, VDT} or {NST, VST} there are 121 combinations of experimental results, we calculated the average accuracy. For instance, to calculate the average accuracy of fixed {NDT, VDT}, we took all the 121 results from the experiments in which their {NDT, VDT} is the fixed {NDT, VDT} and, then averaged them. Figure 6, in which the whiter is the better, presents the accuracy, and the false positive rate of both sets using heat maps. Since many configurations had no false negatives, no false negative analysis is included in the heat maps.

An interesting fact is that the more accurate, on average, a particular instance of {NDT, VDT} or {NST, VST} is, the more affected by the other set the accuracy is. Hence, if a particular {NDT, VDT} configuration had a low average accuracy, the accuracy of a particular configuration {NDT, VDT, NST, VST} with that {NDT, VDT} values would be low independently from the {NST, VST} values. But, if the {NDT, VDT} configuration had a high average accuracy, the accuracy of a particular configuration {NDT, VDT, NST, VST} with that {NDT, VDT} values would strongly depend from the {NST, VST} values. The situation is analogous when analyzing the average accuracy of the {NST, VST} configurations.

From Figure 6a to Figure 6b, it can be seen that when NDT is 5 and VDT is 1, the detector had a good accuracy and not many false positives. In contrast, Figure 6c and Figure 6d evidence that the most restrictive values for NST and VST that had the best accuracy and the lowest false positive rate is 0.1 for both. This configuration also had the best performance among the 14641 tested configurations.

5.1.2 Comparison with traditional cohesive metrics for WSDL documents

As summarized in Section 3, there are several cohesion metrics [27, 35, 36] that base on the implementation source code, but there is a lack of cohesion metrics based on high-level designs. For instance, Lack of Cohesion Method (LCOM) [27] requires accessing the source code, since it is defined as the number of pair of methods that do not share instance variables, minus the number of pair of methods with shared instance variable. Despite of this, our detectors' heuristic is inspired in a well-known automatic document classification one. Taking into account that this is an uncommon way of measuring cohesion, we compared it with an adaptation of the class-level cohesion metric introduced in [36]. The work presented in [36] builds matrixes, or graphs, for denoting similarity between pairs of methods and pairs of attribute types in a class. Then, class cohesion is computed as the overall similarity among the class pairs.

Since WSDL documents contain less information that classes source code, we adapted the idea of using instance variables and methods relationships. Thus, the *methods-methods* graph, which is built by counting the number of methods invocations from within other methods, was omitted. Instead, we adapted this idea by basing on how operations in the same port-type are related to defined XSD data-types. To do this, we constructed a graph for a port-type using the following steps:

1. Add all operations in the port-type as nodes.
2. Add all messages referenced by the operations in the graph as nodes, and the references as edge.
3. Add all complex data-types referenced by messages in the graph as nodes, and the references as edges.
4. Add all complex data-types referenced by other data-types in the graph as nodes, and the references as edges.
5. Repeat step 4, until no node is added.

Notice that this adaptation does not include primitive data-types, such as strings, integers, base64, and array of them, in the graph. Then, to detect whether there is lack of cohesion, our heuristic verifies if the graph is a connected graph, which is similar to how classic cohesion metrics operate. For instance, Tight Class Cohesion (TCC) measures the relative number of directly connected pairs of methods of a class, and Loose Class Cohesion (LCC), which measures the relative number of directly or transitively connected pairs of methods of a class. These two metrics consider two methods to be connected if they share at least one instance variable or one of the methods invokes the other.

To evaluate this adaptation, we used the same methodology and data-set described in Section 5.1. Accordingly, achieved results show that though based on accepted techniques for measuring cohesion, this adapted heuristic missed 40 Low cohesive operations in the same port-type anti-pattern occurrences and incorrectly detected this anti-pattern in 7 WSDL documents, while detected 3 real occurrences of the anti-pattern. This means that the adaptation had an accuracy of 88.01%, a false positive rate of 77.78%, and a false negative rate of 10.44%. Compared to our proposed heuristic, this adaptation has a better accuracy, but the high rate of false positive renders this adaptation ineffective.

This high false positive rate is probably cause by the fact that WSDL documents have less information about the functionality of its related software than the source code. On the other hand, preliminary results in [47] show that classical software metrics can be used to prevent bad practices in WSDL documents when they are generated using the Code-First methodology. This is because with Code-First the WSDL document of a service is derived from its implementation source code, thus classic cohesion metrics can be calculated from it.

5.2 Second Experiment: Al-Masri and Mahmoud's quality metrics and the anti-patterns' detectors

In [22], Al-Masri and Mahmoud surveyed quality metrics from a real world data-set of Web Services, known as QWS [48], which is publicly available[e]. We have calculated the Pearson's correlation between anti-pattern occurrences and the quality metrics. Pearson's correlation was used because we expected to find a direct relationship between Al-Masri and Mahmoud's quality metrics and the anti-patterns. For the correlation analysis, we employed only a subset of this data-set, which consists of 365 Web Services that exist on both [21] and [48] data-sets. The reason to do this was that we have their associated WSDL documents, their quality metrics, and the manual peer-reviewed analysis of anti-patterns occurrences.

The results showed that some of the Al-Masri and Mahmoud's quality metrics have a significant correlation with the anti-patterns, but others have not. However, when we analyzed these results, the metrics that have no correlation are: response time, availability, throughput, successability, reliability, latency, WsRF and class. Since these are technical metrics unrelated to WSDL document quality, it is reasonable that they have no correlation with the anti-patterns, which reflect a WSDL document quality. In contrast, the metrics that have a correlation with the anti-patterns are related to WSDL document quality [22]; these metrics are:

- Compliance: The degree to which a WSDL document grammatically conforms to the WSDL specification.
- Best Practices: The degree to which a Web service complies with WS-I profile guidelines.
- Documentation: The amount of textual documentation in description tags including service, ports, and operations.

In Figure 7, we present the correlation between anti-pattern occurrences and these three quality metrics in the QWS data-set. A correlation value higher than zero means that when one variable rises, the other variable value tends also to rise, while a correlation value lower than zero means that when one variable rises, the other variable value tends to decrease. A correlation value near zero means that the values of the variables are independent, i.e., anti-pattern occurrences and Al-Masri and Mahmoud's

[e] The QWS data-set: `http://www.uoguelph.ca/~qmahmoud/qws/index.html`

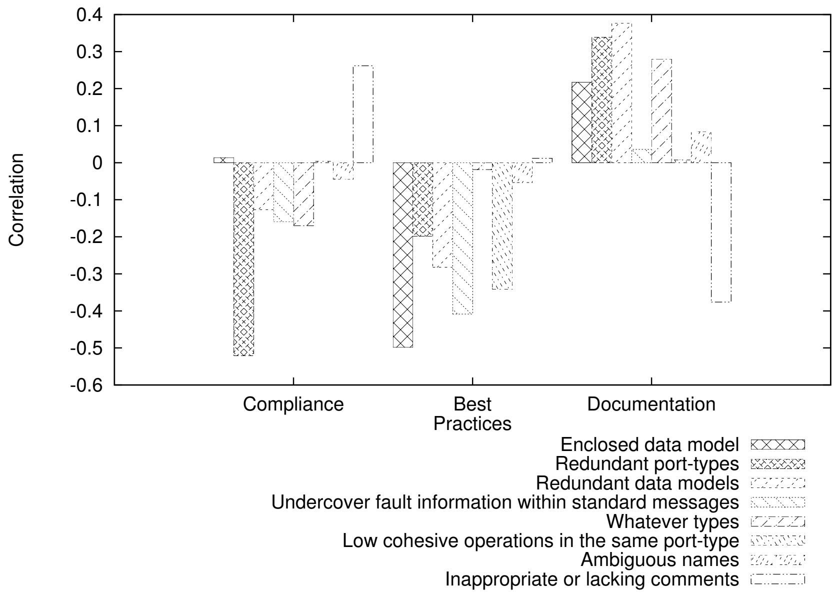


Fig. 7. Correlation between Anti-patterns and the Al-Masri and Mahmoud's quality metrics.

quality metrics are not related. Finally, it is important to notice that correlation values are neither too high nor too low because we are correlating a discrete two-value variable (anti-pattern occurrences) and it cannot have a linear relation to a continuous value.

When using Al-Masri and Mahmoud's quality metrics, low values stand for low quality, and high values stand for high quality. On the other hand, the anti-patterns' variable is zero for not affected WSDL documents and one for affected ones. Therefore, when the correlation between an Al-Masri and Mahmoud's quality metric and an anti-pattern occurrences is negative, it means that having high value in this metric, a WSDL document is unlikely to be affected by the anti-pattern. And when this quality is low, it is very likely that the WSDL document suffers from the anti-pattern. In contrast, a positive correlation means that if a WSDL document has a high value in this metric, it is likely that this WSDL document suffers from the anti-pattern. When the value of the metric is low, the anti-pattern presence is unlikely.

In order to validate the results presented in Figure 7, we calculated the p-value for these correlations. Table 3 presents the p-values for each correlation between an anti-pattern and a metric. The p-value is the possibility that the correlation between two values is zero. This table confirms the finding presented in Figure 7 because the absolute value of the correlation between two values is always less than 0.09 when their p-value is higher than 0.05. In contrast, if the p-value is less than 0.05, the absolute value of correlation between the values is always higher than 0.1. Furthermore, the absolute value of correlation is higher than 0.15 when the p-value is lower than 0.01. Although these are low correlation values, they are clearly separated, and the correlation is higher when the p-value is lower.

According to our results, when the Compliance metric is high in a WSDL document, this document tends not to be affected by most of the anti-patterns. The exceptions to this are the *Enclosed data-model* anti-pattern, for which correlation is near zero, and the *Inappropriate or lacking comments* anti-pattern that is a highly correlated anti-pattern. The first exception is sound because both options,

Table 3. Pearson Correlation p-value

Anti-Pattern	Compliance	Best Practices	Documentation
Enclosed data model	≈ 0.80	< 0.01	< 0.01
Redundant port-types	< 0.01	< 0.01	< 0.01
Redundant data models	< 0.05	< 0.01	< 0.01
Whatever types	< 0.01	< 0.01	≈ 0.50
Undercover fault information within standard messages	< 0.01	≈ 0.72	< 0.01
Low cohesive operations in the same port-type	≈ 0.94	< 0.01	≈ 0.89
Ambiguous names	≈ 0.39	≈ 0.31	≈ 0.11
Inappropriate or lacking comments	< 0.01	≈ 0.82	< 0.01

having enclosed data-model or importing them from an XSD file, are WSDL compliant. On the other hand, the problems with documentation might stem from that when a WSDL document has comment, it is likely that its developer has manually modified it or that the developer has built the WSDL document from scratch, being both error-prone tasks.

Similarly to the Compliance metric, the Best Practices metric has a negative correlation with most of the anti-patterns. The only exceptions are *Whatever types*, *Ambiguous names* and *Inappropriate or lacking comments* anti-patterns. These correlations are near zero, i.e., this metric and these anti-pattern occurrences are unrelated. This is probably because the Best Practices metric is related to WS-I guidelines that aim to improve the technical interoperability, but not the usability of a Web Service; and these anti-patterns are precisely connected to WSDL document readability and understandability, but a WSDL document with symptoms of these anti-patterns can be as good or as bad as a WSDL document without symptoms of these anti-patterns from the WS-I profile point of view.

Finally, the Documentation metric, which represents the percentage of elements in a WSDL document that contain comments, is only correlated negatively to the *Inappropriate or lacking comments* anti-pattern. This is expected because if a developer introduces comments in a WSDL document, they are intended to be read. The other anti-patterns are either not correlated or have a positive correlation. The positive correlation might result from errors that developers made when they edited WSDL documents. This is consistent with the observation made on the correlations of Compliance and Best Practices metrics.

All in all, while other anti-pattern occurrences tend to decrease or are not affected when Compliance and Best Practices metrics value arises, *Inappropriate or lacking comments* anti-pattern occurrences tend to increase. In addition, *Inappropriate or lacking comments* anti-pattern occurrences are less frequent when the Documentation metric value is high, while the other anti-patterns have the opposite behavior. We consider these findings of great interest because comments have been proved to be essential for both service registries, and service consumers [11], thereby we present a deeper analysis the correlation between *Inappropriate or lacking comments* anti-pattern and Al-Masri and Mahmoud's quality metrics in the next section.

Table 4. Statistical information.

Statistical value/Metric	Compliance	Best Practices	Documentation	WsRF	Class
Average Value	83.709	80.671	47.292	66.649	2.781
Standard deviation	8.773	6.695	36.309	11.506	0.980
Correlation	-0.262	-0.007	0.375	0.207	-0.188
Average with the anti-pattern	85.270	80.705	38.052	65.032	2.907
Average without the anti-pattern	80.315	80.596	67.394	70.166	2.508

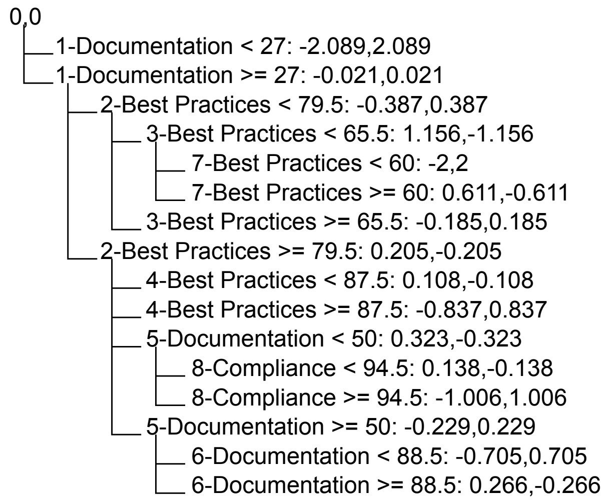
5.3 *Third Experiment: Further analysis of* Inappropriate or lacking comments *anti-pattern detection and Al-Masri and Mahmoud's quality metrics*

We have performed a deeper analysis of the *Inappropriate or lacking comments* anti-pattern and the Al-Masri and Mahmoud's quality metrics. Aside from comments importance, the Al-Masri and Mahmoud's quality metrics correlation with the *Inappropriate or lacking comments* anti-pattern is almost the opposite of the correlation with other anti-patterns. Therefore, the goal of this section is to make a detailed analysis of the correlation between the *Inappropriate or lacking comments* anti-pattern in a WSDL document and the quality values assigned in Al-Masri and Mahmoud work [22]. This analysis was not limited to the Pearson's product-moment correlation coefficient like in the previous section. We have also used several statistical indicators to study the relationship between this anti-pattern detector output and the quality metrics. In addition to analyzing new indicators, we also added the following quality metrics to the analysis:

- WsRF: Web Service Relevancy Function is a rank for Web Service Quality [48].
- Class: a level representing service offering qualities. This metric accepts a discrete value from 1 to 4, but in this case, a lower value stands for better quality.

Table 4 presents the statistical data obtained when the WSDL documents were analyzed. The most related value is the Documentation metric, whose correlation is 0.37, and this is reflected in its average value when the anti-pattern is present or not. The other attribute that presents a positive correlation is WsRF, although it is less than the Documentation metric. However, it is important to notice that, by definition, the WsRF value depends on the values of the other metrics. Therefore, its correlation might be a result of the Documentation metric correlation. The Compliance metric is also related to the presence of this anti-pattern, but in a negative manner. This means that the more compliant the WSDL document is, the more likely the *Inappropriate or lacking comments* anti-pattern occurrence is. To confirm that this correlation is not due to a characteristic of our detector, the correlation between the Compliance metric and the Documentation metric was calculated. It was -0.28, which confirms that Compliance is in detriment of Documentation and vice versa, supporting our hypothesis. Finally, as a lower value means good quality, the negative correlation represents exactly the same as the positive correlation for the WsRF metric.

Based on these correlations, we also analyzed whether it is possible to predict this anti-pattern occurrences based on these attributes' values. This analysis was performed using decision trees trained using Class, Documentation, Best practices and Compliance attributes as inputs and the anti-pattern detector result as output. The selected algorithm to generate this tree was Multiclass alternating deci-



```
0,0
|__1-Documentation < 27: -2.089,2.089
|__1-Documentation >= 27: -0.021,0.021
   |__2-Best Practices < 79.5: -0.387,0.387
   |  |__3-Best Practices < 65.5: 1.156,-1.156
   |  |  |__7-Best Practices < 60: -2,2
   |  |  |__7-Best Practices >= 60: 0.611,-0.611
   |  |__3-Best Practices >= 65.5: -0.185,0.185
   |__2-Best Practices >= 79.5: 0.205,-0.205
      |__4-Best Practices < 87.5: 0.108,-0.108
      |__4-Best Practices >= 87.5: -0.837,0.837
      |__5-Documentation < 50: 0.323,-0.323
      |  |__8-Compliance < 94.5: 0.138,-0.138
      |  |__8-Compliance >= 94.5: -1.006,1.006
      |__5-Documentation >= 50: -0.229,0.229
         |__6-Documentation < 88.5: -0.705,0.705
         |__6-Documentation >= 88.5: 0.266,-0.266
```

Fig. 8. Decision tree.

sion trees [49] because it was the most accurate decision tree among several techniques, provided by Weka [50], for this problem.

Figure 8 depicts the resulting decision tree. To use this tree with an instance, it is necessary an auxiliary vector initialized as the root node, i.e., (0,0). Then, the algorithm follows the tree to add the values in all the nodes that the current instance fits to the auxiliary vector. If the first number is the greater, the tree predicts that the anti-pattern is not present, otherwise the prediction is that the anti-pattern is present.

The tree predicts that if the Documentation metric value is less than 27% the anti-pattern will be detected independently of the other metrics. This makes senses because a low value in this metric represents that the WSDL document has little comments. In contrast, the Best practice metric behaves in a not homogeneous way because if it is in some ranges it suggests that the anti-pattern will not be detected, but when the attribute takes extreme values, the decision tree suggests that the anti-pattern will be detected. A final observation, when the Documentation metric value is between 27% and 50%, and the Best Practice is more than 79.5% (condition that meets 17% of the data-set instances), then a high Compliance value (more than 94.5%) suggests that the anti-pattern will be detected.

6 Future Research Possibilities

Automatically detecting anti-patterns in WSDL documents is the previous step to assist developers in refactoring their WSDL documents using software tools. We believe that this kind of tools will make the anti-pattern catalog even more valuable because they will ensure that the refactoring guidelines will be properly applied. Therefore, an open research question is how to automatize WSDL document refactorings.

The anti-pattern refactorings presented in [11] assume that service developers are actually involved in the WSDL document construction, i.e. they employ the *contract-first* approach. However, there is another way of create WSDL documents called *code-first*, which means that developers first write a service implementation and then generates the corresponding service contract by automatically extracting and deriving the interface from the implemented code. This means that WSDL documents are not directly created by humans, but are automatically derived from programming languages via language-dependent tools. As a result, the generated WSDL documents are not verified by service developers. Therefore, another research question comprises analyzing whether these techniques could work along with code-first WSDL document generation tools, informing potential errors in the code from which WSDL documents are derived.

Similar to the first open question, it has been shown in [47] that performing common Fowler et al.'s refactorings in the code implementing code-first Web Services, may prevent WSDL documents of having anti-patterns occurrences.

Since many of the presented heuristics are language dependent, we will evaluate their performance in other languages and incorporating further sources of data, such as Web pages pointing to services descriptions. Since Web pages are the typical manner of describing RESTful services, we will also research on potential issues discoverability issues in RESTful service descriptions and how to detect them. In addition, we are evaluating other techniques, e.g. clustering, to detect *Low cohesive operations within the same port-type* anti-pattern without the necessity of having a previously classified data-set. Finally, another extension is to combine the detectors with service registries to automatically mitigate the anti-pattern effects by, for instance, eliminating repeated port-types or not indexing redundant information.

7 Conclusions

In previous works [11, 13] the implications of the use of poorly written WSDL documents against discovery and human discoverers' understandability have been empirically proven. Broadly, previous works highlight the importance of offering self-explanatory WSDL documents, mostly because discoverable and understandable services potentially mean more applications that re-use them [51]. For paid Web Services, this means more incomes. This paper presents novel algorithms and heuristics for detecting recurrent discoverability problems in WSDL documents. These algorithms and heuristics have been implemented and in turn employed for detecting such problems in a corpus of real-world Web Services, which had been peer reviewed by humans. The results reported in this paper show that the averaged accuracy of the proposed detectors was 93.14%, and the false positive and false negative rates of 4.064% and 9.911%, respectively. It is worth noting that these results are data set specific and may vary with another data set, mostly for the heuristics accuracy, though the size of the employed corpus is to some extent representative.

Two additional experiments were conducted to further test the precision of one proposed heuristic. This heuristic deals with detecting the anti-pattern that has the strongest impact on discovery, as reported by [11], namely *Inappropriate or lacking comments*. These experiments correlated heuristic results with Al-Masri and Mahmoud's study about Web Services quality shown in [22]. Accordingly, the statistical correlation analysis provides more empirical evidence about the precision of the associated heuristic.

To conclude, evaluation results empirically confirm that the proposed detectors can minimize the impact of the commonest bad practices by helping developers to identify potential problems in their services before they are made available. All in all, in order to materialize the vision of a global market of interoperable and discoverable Web Services more effort should be placed on answering the research question related to preventing anti-patterns on code-first WSDL documents and automatically refactoring contract-first ones.

Acknowledgments

We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311.

References

1. Paul Grefen, Heiko Ludwig, Asit Dan, and Samuil Angelov. An analysis of Web Services support for dynamic business process outsourcing. *Information and Software Technology*, 48(11):1115 – 1134, 2006.
2. Yi Wei and M.B. Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72 –75, nov.-dec. 2010.
3. Chun-Lung Huang, Chi-Chun Lo, Kuo-Ming Chao, and Muhammad Younas. Reaching consensus: A moderated fuzzy Web Services discovery method. *Information and Software Technology*, 48(6):410 – 423, 2006.
4. Mohsen Sharifi, Somayeh Bakhtiari Ramezani, and Amin Amirlatifi. Predictive self-healing of web services using health score. *Journal Web Engineering*, 11(1):79–92, 2012.
5. Marco Crasso, Alejandro Zunino, and Marcelo Campo. A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management*, 22:103–134, 2011.
6. David Martin, Mark Burstein, Drew Mcdermott, Sheila Mcilraith, Massimo Paolucci, Katia Sycara, Deborah L. Mcguinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to Web Services with owl-s. *World Wide Web*, 10(3):243–277, 2007.
7. Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for Web Services. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 372–383, Toronto, Canada, August 31 - September 3 2004. Morgan Kaufmann.
8. Marco Crasso, Alejandro Zunino, and Marcelo Campo. Combining query-by-example and query expansion for simplifying Web Service discovery. *Information Systems Frontiers*, in press, 2009.
9. Eleni Stroulia and Yiqiao Wang. Structural and semantic matching for assessing Web Service similarity. *International Journal of Cooperative Information Systems*, 14(4):407–438, June 2005.
10. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Extending movilog for supporting Web Services. *Computer Languages, Systems & Structures*, 33(1):11 – 31, 2007.
11. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001 – 1021, 2010.
12. Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Jack, and Brad A. Myers. Usability challenges for enterprise service-oriented architecture APIs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 193–196, Sept. 2008.
13. M. Brian Blake and Michael F. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12(5):62–69, 2008.
14. Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32, 2005.
15. J. Pasley. Avoid XML schema wildcards for Web Service interfaces. *IEEE Internet Computing*, 10(3):72–79, May-June 2006.
16. Cristian Mateos, Marco Crasso, Juan M. Rodriguez, Alejandro Zunino, and Marcelo Campo. Measuring the impact of the approach to migration in the quality of Web Service interfaces. *Enterprise Information Systems*, in press(0):1–28, 2012.
17. Marco Crasso, Alejandro Zunino, and Marcelo Campo. AWSC: An approach to Web Service classification based on machine learning techniques. *Revista Iberoamericana de Inteligencia Artificial*, 37(12):25–36, 2008.
18. Hinrich Schütze Christopher D. Manning. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
19. Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL'03)*, pages 423–430, 2003.
20. Christopher D. Manning Dan Klein. Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, 2003.
21. Andreas Heß, Eddie Johnston, and Nicholas Kushmerick. ASSAM: A tool for semi-automatically annotating semantic Web Services. In Sheila A.McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors,

International Semantic Web Conference, volume 3298 of *Lecture Notes in Computer Science (LNCS)*, pages 320–334, Hiroshima, Japan, November 7-11 2004. Springer.

22. Eyhab Al-Masri and Qusay H. Mahmoud. WSB: A broker-centric framework for quality-driven web service discovery. *Software: Practice and Experience*, 40:917–941, September 2010.
23. T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
24. Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
25. Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649 – 660, 2008. Software Process and Product Measurement.
26. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485 –496, july-aug. 2008.
27. V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751 –761, oct 1996.
28. Cara Stein, Glenn Cox, and Letha Etzkorn. Exploring the relationship between cohesion and complexity. *Journal of Computer Science*, 1(2):137 –144, 2005.
29. Ninus Khamis, RenÃ© Witte, and Juergen Rilling. Automatic quality assessment of source code comments: The javadocminer. In Christina Hopfe, Yacine Rezgui, Elisabeth MÃ©tais, Alun Preece, and Haijiang Li, editors, *Natural Language Processing and Information Systems*, volume 6177 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2010.
30. Rudolph Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32(3):221–233, 1948.
31. V. Arnaoudova. Improving source code quality through the definition of linguistic antipatterns. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 285 –288, oct. 2010.
32. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. *Software Maintenance and Reengineering, European Conference on*, 0:156–165, 2010.
33. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining java class naming conventions. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM ’11, pages 93–102, Washington, DC, USA, 2011. IEEE Computer Society.
34. Henrik Leopold, Sergey Smirnov, and Jan Mendling. On the refactoring of activity labels in business process models. *Information Systems*, 37(5):443 – 459, 2012.
35. Heung Seok Chae, Yong Rae Kwon, and Doo Hwan Bae. A cohesion measure for object-oriented classes. *Software: Practice and Experience*, 30(12):1405–1431, 2000.
36. Jehad Al Dallal and Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52(12):1346 – 1361, 2010.
37. T. Suzumura, T. Takase, and M. Tatsubori. Optimizing web services performance by differential deserialization. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 185 – 192 vol.1, july 2005.
38. Abdelkarim Erradi and Piyush Maheshwari. A broker-based approach for improving web services reliability. *Web Services, IEEE International Conference on*, 0:355–362, 2005.
39. Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
40. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, July 2008.
41. Eric E. Allen and Robert Cartwright. Safe instantiation in generic java. *Science of Computer Programming*, 59(1-2):26 – 37, 2006. Special Issue on Principles and Practices of Programming in Java (PPPJ 2004).
42. Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.
43. J. Sanger R. Feldman. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2006.
44. M. F. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14:130–137, 1980.
45. Nicola Stokes. *Aplications of lexical cohesion analysis in the topic detection and tracking domain.* PhD

thesis, University College Dublin, 2004.

46. Sanda Harabagiu and Dan Moldovan. *Natural language processing and knowledge representation*, chapter Enriching the WordNet taxonomy with contextual knowledge acquiered from text, pages 301–334. MIT Press, 2000.
47. José Luis Ordiales Coscia, Cristian Mateos, Marco Crasso, and Alejandro Zunino. Avoiding WSDL Bad Practices in Code-First Web Services. In *Proceedings of the 12th Argentine Symposium on Software Engineering (ASSE2011) - 40th JAIIO*, pages 1–12, 2011.
48. Eyhab Al-Masri; Qusay H. Mahmoud. Qos-based discovery and ranking of Web Services. In *Proceedings of the 16th International Conference on Computer Communications and Networks (ICCCN'07)*, pages 529–534, 2007.
49. Geoffrey Holmes, Bernhard Pfahringer, Richard Kirkby, Eibe Frank, and Mark Hall. Multiclass alternating decision trees. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *Machine Learning: ECML 2002*, volume 2430 of *Lecture Notes in Computer Science*, pages 105–122. Springer Berlin / Heidelberg, 2002.
50. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.
51. J.L. Pastrana, E. Pimentel, and M. Katrib. Qos-enabled and self-adaptive connectors for Web Services composition and coordination. *Computer Languages, Systems & Structures*, 37(1):2 – 23, 2011.