Automatically detecting opportunities for Web Service descriptions improvement

Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, Marcelo Campo

ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682 ext. 35. Fax.: +54 (2293) 439683
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Abstract. Mostly *e*-business and *e*-applications rely on the Service Oriented Computing paradigm and its most popular implementation, namely Web Services. When properly implemented and described, Web Services can be dynamically discovered and reused using Internet technologies, pushing interoperability to unprecedented levels. However, poorly described Web Services are rather difficult to be discovered, understood, and reused. This paper presents heuristics for automatically detecting common pitfalls that should be avoided when creating Web Service descriptions. Experimental results with ca. 400 real-world Web Services, empirically show the feasibility of the proposed heuristics.

Keywords: Web Service modeling, Web Service discoverability anti-patterns

1 Introduction

The success encountered by the Internet encourages practitioners, companies and governments to create software that utilizes information and services that third-parties have made public in the Web. This, besides encouraging the aforementioned actors to offer their information and services in the same way, spreads into a new kind of software, namely *e*-applications, *e*-business, and *e*-government [1]. Nowadays, the *Service Oriented Computing* (SOC) paradigm [2] is used for developing this kind of software.

With SOC, software development involves a service provider, who offers services and advertises them in a public registry, and service consumers, who use such a registry to find the services that they need [3]. Using open standards, such as SOAP, HTTP, and XML, to implement the SOC paradigm is by now commonplace in the software industry because these standards allow the integration of different pieces of software independently of their platform and location. Basically, providers describe their services using Web Service Description Language (WSDL), which is an XML-based language, and advertise them using Universal Description, Discovery and Integration (UDDI), which uses XML for representing services meta-data and SOAP for querying it. The term Web Services refers to these standards that support SOC across the Web [4].

SOC and Web Services have been broadly embraced by the software industry. The ever-growing number of publicly available services represents “either more freedom or more chaos for service consumers” [5], mainly because of the limited search capabilities of UDDI, the incorrect usage of standards to describe Web Services, and that they do not consider the semantics of services in an explicit and non ambiguous

way [6]. Several registry enhancements have been proposed to improve the experience of service consumers. The approach to service discovery that bases on exploiting every possible piece of information conveyed in standard service descriptions, has shown favorable outcomes [7]. This approach bases on the fact that when WSDL documents are well-written the signatures and associated comments of their offered operations, convey keywords relevant to index the services [7]. Well-written WSDL documents are essential for not only such approach, but also service consumers because if they do not understand what a service does, they would not select the service. Unfortunately, despite such importance and Foster's words "Web Services have little value if other cannot discover, access, and make sense of them" [8], it seems that providers tend not to care about Web Services discoverability and understandability, as pointed by [9,10,11].

In our previous work [10], we introduced a catalog of WSDL-based Web Services discoverability anti-patterns. Besides measuring the impact of each anti-pattern on discovery, the study assesses the implications of anti-patterns on users' ability to make sense of WSDL documents. The catalog consists of eight anti-patterns having a name, a problem description, and a soundly refactor procedure. However the results of the study motivate anti-patterns refactoring, manually looking for an anti-pattern in WSDL documents might be a time consuming and complex task. Thus, this paper presents heuristics to automatically detect the anti-patterns of the catalog. These heuristics have been experimentally validated with a real-world data-set, showing an averaged accuracy of 98.5%. Therefore, the main contribution of this paper is:

– the definition and validation of novel heuristics for automatically detecting anti-patterns that already have been proven to be opportunities for improving the discoverability and understandability of Web Services in [10].

The rest of the paper is organized as follows: Sect. 2 explains the essential characteristics of the WSDL, the service discovery process, and the Web Services discoverability anti-patterns, Sect. 3 explains the proposed heuristics, while Sect. 4 reports conducted experiments, finally Sect. 5 presents conclusions and opportunities of future research.

2 Background

WSDL is an XML-based language that allows providers to describe the service functionality as a set of *port-types*. A port-type arranges different *operations* whose invocation is based on exchanging *messages*: one message with input data, other with the result, and another with error information, optionally. Port-types, operations and messages, must be named with unique names. Messages consist of *parts* that are arranged according to specific data-types defined using the XML Schema Definition (XSD) language. XSD offers constructors for defining simple types (e.g., integer and string), and more elaborate mechanisms for defining complex elements. Data-type definitions can be put into a WSDL document or into a separate file and imported from any WSDL document afterward. The grammar of the WSDL[1] can be summarized as follows:

[1] Note that "?" means optional and "*" means none or many.

```
<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>
```

UDDI[2] was originally defined as the discovery protocol of Web Services. However, UDDI has been proven to be an ineffective discovery method for large registries because it is a keyword based discovery method [12]. As a result two approaches to discovery have been taken, the first one relies on an extended WSDL to include ontology based descriptions of the service. Although this approach allows automatic service discovery, the industry has not adopted because the high effort required for implementing it [13]. The second direction is applying information retrieval (IR) techniques to WSDL documents for allowing search Web Service [7]. The main strength of using IR is that there is no need of modifying existent WSDL documents. However, this is also a drawback because many of those documents are not well-written.

Although the importance of well-designed WSDL document has been identified as a central concern in Web Service reuse [14,15], different studies have pointed out the existence of widespread problems in documentation [16], naming [11] and interface design [9,17] in real-life WSDL documents.

In [10], we have explicitly addressed the quality of WSDL documents from the perspective of a discoverer, pursuing recurrent problems that attempt against the understandability and discoverability of services. To do this, we have studied publicly available WSDL documents looking for common problems. As a result, the study presents a catalog of bad practices that frequently occur in the analyzed corpus, along with hints about how to detect problem symptoms, and refactoring guidelines to solve them. Specifically, each observed bad practice has been described in a general way that includes a description of: the problem, its solution, and an illustrative example, thus we refer to the catalog as a catalog of Web Services discoverability *anti-patterns* [18].

Table 1 summarizes the identified anti-patterns. The “Symptoms” column describes the bad practice associated with an anti-pattern. The column named “Manifests” presents a classification based on how an anti-pattern can be detected. Anti-pattern manifestation can take three values: *Evident*, *Not immediately apparent*, and *Present in service*

[2] UDDI `http://uddi.xml.org/`

Table 1: Catalog of Web Service discoverability anti-patterns.

Anti-pattern	Symptoms	Manifest
Enclosed data model	Occurs when the data-type definitions are placed in WSDL documents rather than in separate XSD ones.	Evident
Redundant port-types	Occurs when port-types offer the same set of operations.	Evident
Redundant data models	Occurs when many data-types for representing the same objects of the problem domain coexist in a WSDL document.	Evident
Whatever types	Occurs when a data-type represents any object of the domain.	Evident
Inappropriate or lacking comments	Occurs when (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	Occurs when port-types have weak semantic cohesion.	Not immediately apparent
Ambiguous names	Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent
Undercover fault information within standard messages	Occurs when output messages are used to notify service errors. Sometimes (1) whatever types are returned and operation comments suggest anti-pattern occurrence. Otherwise (2) it is necessary to analyze service implementation.	(1) Not immediately apparent, or (2) Present in service implementation

implementation. An anti-pattern is *Evident* if it can be detected by analyzing only the structure, or syntax of the WSDL document. *Not immediately apparent* means that detecting the anti-pattern requires a semantic analysis as well. Finally, *Present in service implementation* anti-patterns may not show themselves in the WSDL document, thus requiring the execution of the associated service to be detected. This classification drives the approach to detect an anti-pattern, as will be explained in next section. It is worth noting that two anti-patterns are classified in many categories, and that the proposed heuristics detect them when they manifest according to their 1^{st} classification.

3 Automatic discoverability anti-pattern detection

Our anti-pattern detection approach bases on an incremental process, in which a WSDL document is passed through eight different heuristics. Each heuristic deals with the detection of a particular anti-pattern. The idea of having individual detection heuristics stems from the fact that anti-pattern occurrences are mutually independent [10]. In other words, the occurrence of one anti-pattern does not imply the occurrence of another one. Below, the heuristics are discussed regarding the way each anti-pattern manifests.

3.1 Evident anti-patterns

The detection of Evident anti-patterns is based on rules, which are applied to the WSDL document grammar, because the manifestation of these anti-patterns is syntactical.

Algorithm 1 Heuristic for detecting *Repeated data model anti-pattern*.

```
1:  function REDUNDANT(element1, element2)        ▷ The first time recives two data definitions
2:      if !HAVESAMEATTRIBUTES(element1, element2) then
3:          return false
4:      end if
5:      chidren1 ← GETCHILDREN(element1)
6:      chidren2 ← GETCHILDREN(element2)
7:      if SIZE(children1)! = SIZE(children2) then
8:          return false
9:      end if
10:     for i ← 0; i < SIZE(children1); i + + do
11:         child1 ← children1[i]
12:         child2 ← children2[i]
13:         if !REDUNDANT(child1, child2) then
14:             return false
15:         end if
16:     end for
17:     return true
18: end function
```

To detect *Enclosed data model* anti-pattern, it is necessary to know if the data-types exchanged by service operations are defined in the WSDL document or imported from some where else. To do this, our heuristic checks if the `<types>` tag is present in a WSDL document. If it is not present, the data model is not defined in the WSDL document; therefore, the anti-pattern is not present. Instead, when the WSDL document contains the `<types>` tag, the heuristic analyzes whether the tag is empty or it contains one or more `<schema>` tags. If the `<types>` tag is empty, it again means that no type is defined, which suggests that the anti-pattern is not present. On the other hand, 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 detection of *Redundant port-types* anti-pattern requires to revise if a port-type is defined several times in the same WSDL document. Usually, to support another invocation method for a service, providers tend to repeat a port-type, but using data-types specially designed for the new invocation method [10]. Consequently, it is not possible to detect anti-pattern occurrences by looking for exact matching between two port-types. Therefore, the heuristic analyzes if two port-types have the same number of operations and if they have the same names, skipping any message similarity checks.

Redundant data model anti-pattern manifestation is different from *Redundant port-type* anti-pattern manifestation. With *Redundant data model* anti-pattern, the names of the elements that describe the data-type are likely to change, but not the structure of the data-type. Thus, the detection of this anti-pattern involves comparing the structure of each defined data-type. Algorithm 1 shows how two data-type definitions are compared.

The two commonest forms in which the *Whatever type* anti-pattern manifests itself are: (1) when a data-type is defined using the XSD primitive type “anyType”, (2) when a data-type definition includes the `<any>` tag. Both cases allow developers to leave a data-type part undefined because any valid XML content can be inserted afterward in

such an undefined part. Therefore, the corresponding heuristic analyzes if <any> tag is present, or some tag have "anyType" as a value of its "type" property.

Finally, another heuristic checks that that all operations within a WSDL document have associated the <documentation> tag and its content is not empty, otherwise an evident occurrence of *Inappropriate or lacking comments* anti-pattern is present.

3.2 Not immediately apparent anti-patterns

Not immediately apparent anti-patterns cannot be detected by syntactically analyzing the WSDL grammar. Instead, their detection requires analyzing the semantics of comments and names present in WSDL documents [10]. Therefore, the rules for detecting anti-patterns of this group are more complex than those of the previous section.

In order to detect if the *Low cohesive operations in the same port-type* anti-pattern occurs in a service description, it is necessary to verify that port-type operations belong to the same domain. Broadly, the heuristic aims to deduce the domain of each individual operation, and then compares deduced domains looking for mismatches. Since the information available of the operations are their names, comments, messages and datatypes, which are textual information, our heuristic reduces the problem of classifying operations according to their domain to the well-known problem of classifying text.

Current implementation of the heuristic employs a variation of Rocchio classifier, called Rocchio TF-IDF, because a previous work [19] has empirically shown that Rocchio TF-IDF outperformed other classifiers for the Web Services context. Rocchio TF-IDF represents textual information as vectors, in which each dimension stands for a term and its magnitude is the weight of the term related with the text. Having represented all the textual information of a domain as vectors, the average vector, called centroid, is built for representing the domain. Then, the domain of an operation is deduced by representing it as a vector and comparing it to each domain centroid. Finally, the domain of the most similar average vector is returned as being the domain of that operation. For the sake of conciseness, the reader asking for deep explanations and details about gathering textual information from WSDL documents, representing it as vectors, centroid construction and similarity calculations should refer to [19].

Using Rocchio TF-IDF, operations can be easily classified and if a port-type contains operations that belong to different domains the anti-pattern is considered to be present. The main disadvantages of using Rocchio TF-IDF are that the classifier is only able to classify operations in known domains, and requires an expert to classify a training-set of operations according with their domain. Thus, for the experiments we employed a corpus of WSDL documents that have been previously classified.

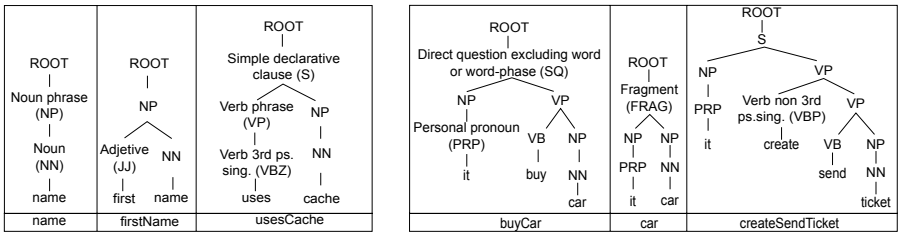
The *Ambiguous names* anti-pattern is another Not immediately apparent anti-pattern, which deals with non-explanatory WSDL element names. The first step to detect naming problems in a WSDL document is to check whether the length of any name is neither too short nor too long. Thus, the associated heuristic checks if the length of any name is between a fixed range of characters, otherwise the name is considered as an occurrence of the anti-pattern. For the experiments, we set the range in [3:30] characters.

Second, several words have been identified to be related with non-explanatory or too general names [11]. The unrecommended words are: *thing, class, param, arg, obj, some, execute, return, body, foo, http, soap, result, input, output, in, out.* A name that

has any of these words probably is too general; therefore if a name contains one of these words, the corresponding heuristic detects it as an ambiguous name.

Third, each name should have an adequate grammatical structure. The name of an operation should be in the form: <verb> "+" {<noun>|<noun phrase>} because an operation is an action, but in the case of exchanged data (e.g. a message part), its name should be a {<noun>|<noun phrase>} because it represents a concept [10].

To grammatically analyze the structure of a name, our heuristic applies a probabilistic context free grammar parser [20] to operation and part names. With this kind of parsers, a sentence is analyzed and associated with rules that form one or more parsing trees as in traditional context free grammar, but each rule has an independent probability. Therefore, it is possible to calculate the most probably parsing tree for a given sentence by multiplying the probability of all the rules of each derived parsing tree.



(a) Parsing message part names (b) Parsing operation names

Fig. 1: Parsing tree examples.

The heuristic to analyze a message name is to check if the parsing tree derived by the parser has no verb tags. For example, Fig. 1a depicts the parsing tree of three message names: "name", "firstName", and "usesCache". The first and second names are correct, because their parsing trees do not contain verbs. However, the name "usesCache", starts with a verb so it represents an action and thus it is not correct.

When analyzing an operation name, the heuristic adds to the operation name the word "it" at the beginning of the name to indicate the noun that it should be missing in the name of an operation. 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 closed enough to the correct sentence because having probabilistic rules makes the parser able to handle malformed sentences [20]. Then, our heuristic counts the number of verbs in the parsing tree. If the number of verbs is different from one, the *Ambiguous name* anti-pattern is detected. Figure 1b 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 one and only one action. In contrast, the second name, which is "car", is a noun then 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 it has two verbs meaning that the operation actually performs two actions.

Finally, the last heuristic aims at detecting Not immediately apparent occurrences of *Undercover fault information within standard messages* anti-pattern. Commonly, this anti-pattern has footprints in WSDL documents, but sometimes it requires to analyze service implementation. Our heuristic only detects this anti-pattern when is the first case. To do this, the heuristic verifies whether an operation has a `<fault>` message defined that means that the errors are handled in the correct manner [10]. 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 an occurrence of the Whatever type anti-pattern in the output, and keywords in the operation comments that indicate the presence of the anti-pattern. The set of keywords is: *fault, error, fail, overflow, exception, stackTrace*.

4 Experimental evaluation

Previous section describes the proposed heuristics for automatically detecting the discoverability anti-patterns introduced by [10]. This section describes the experiments that have been conducted to evaluate the detection effectiveness of these heuristics.

The followed evaluation methodology involves manually analyzing each WSDL document to identify the anti-patterns it has, peer-reviewing manual results (at least three different people reviewed each WSDL document), automatically analyzing the WSDL document using the proposed heuristics, and finally comparing both manual and automatic results. These results are 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 heuristic accurately operates for that document. Achieved results are shown using a confusion matrix. Each row of the matrix represents the number of WSDL documents that were automatically classified using the heuristic associated with a particular anti-pattern. The columns of the matrix show manual classifications, i.e. the number of WSDL documents that has the anti-pattern actually.

The described methodology was followed using the data-set of 392 WSDL documents that it is described in [10]. This data-set[3], which was gathered by Hess et al. [21], has been selected because it is an snapshot of publicly available Web Service on Internet. Once each heuristic was fed with the data-set and its results computed, we built the confusion matrixes. Then, we assessed the accuracy, and false positive/negative rates for each matrix. Table 2 shows the confusion matrixes.

The accuracy of each heuristic was calculated as the number of classification matching over the total of analyzed WSDL documents. For instance, the accuracy of the Redundant data model heuristic was $\frac{221+166}{221+2+3+166} = 0.987$. The heuristic for detecting Low cohesive operations within the same port-type anti-pattern achieved the lowest accuracy: 0.775. This could be caused by errors that the classifier introduced. Nevertheless, the averaged accuracy for all heuristics was 0.958.

The false positive rate is the proportion of WSDL documents that a heuristic wrongly labels them as having the corresponding anti-pattern. At the same time, the false negative rate is the proportion of WSDL documents that a heuristic wrongly labels as not

[3] Data-set: `http://www.andreas-hess.info/projects/annotator/index.html`

Table 2: Confusion matrixes for the detection of anti-patterns.

Automatic detection results per anti-pattern		**Manual detection results**	
		Negative	Positive
Enclosed data model	Negative	116	6
	Positive	0	270
Redundant port-types	Negative	161	4
	Positive	0	227
Redundant data models	Negative	221	2
	Positive	3	166
Whatever types	Negative	339	0
	Positive	3	50
Lacking comments	Negative	135	0
	Positive	0	257
Low cohesive operations in the same port-type	Negative	272	10
	Positive	78	32
Ambiguous names	Negative	67	0
	Positive	9	316
Undercover fault information within standard messages	Negative	351	3
	Positive	4	34

having the corresponding anti-pattern. A false negative rate equals to 1 would mean that a detection heuristic missed all anti-pattern occurrences. For these rates, the lower the achieved values the better the detection effectiveness. The averaged false positive rate was 0.036, and the averaged false negative rate was 0.052.

We individually analyzed each WSDL document that was wrongly classified by the heuristics. Afterward, we detected that the reason behind 16 incorrect classifications was that those WSDL documents adhere to the 2001 WSDL standard, whereas the implementation of the heuristics depends on the 1999 standard. Therefore, 16 mismatches were caused by current implementation of the heuristics and not by a heuristic itself.

5 Conclusions and Future Work

Many Web Service problems for being discovered and re-used, have been recognized as having their roots in WSDL discoverability anti-patterns [10]. This paper presents novel heuristics for detecting these anti-patterns. Proposed heuristics have been employed for analyzing a corpus of real-world Web Services, which had been manually analyzed. Reported experiments show that the averaged accuracy of the heuristics was 0.958, and the false positive and false negative averaged rates of 0.036 and 0.052, respectively. All in all, the proposed heuristics represent an advance in the creation of Web Services that are easier to be understood and discovered, which by themselves symbolize the basic

blocks for *e*-applications relying on the SOC paradigm or the new wave of service-oriented software systems, such as Cloud Computing [22], SaaS or PaaS [23].

The anti-pattern detector can minimized the impact of the commonest bad practice by helping developers to detect potential problems in their services before they are made available. In addition, Web Service registries may use the anti-pattern detector for informing service developers about possible problems in their services, so developers can be aware of those problems for avoiding them in future versions of the services.

More experiments should be done in the future, since the reported results can not be generalized, in particular those related to Not immediately apparent anti-patterns. In this sense, we are planning to employ the heuristics with a recently published repository of real Web Services [24]. Besides, this work will be extended to incorporate a heuristic for analyzing the descriptiveness of comments present in WSDL documents. Currently, we are evaluating a heuristic that combines WordNet, an electronic lexical database, and a natural language parser. Preliminary results are encouraging.

Another line of future research involves the synchronization between changes in WSDL documents and service implementations, because removing the identified anti-patterns from a service description may imply changes in the underlying software. Furthermore, some version support technique is necessary to allow consumers that use the old WSDL document version to continue using the service until they migrate to the improved WSDL document [25].

Acknowledgments

We thank Pablo Inchausti and Daniel Molero for helping us to implement the heuristics. Also, thanks to ANPCyT for supporting this research through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

1. Adegboyega Ojo Tomasz Janowski and Elsa Estevez. Rapid development of electronic public services: a case study in electronic licensing service. In *Proceedings of the 8th annual international conference on Digital government research (DG.O'07)*, pages 292–293, 2007.
2. Martin Bichler and Kwei-Jay Lin. Service-Oriented Computing. *Computer*, 39(3):99–101, 2006.
3. Michael P. Papazoglou and Willem-Jan Van Den Heuvel. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2(4):412–442, 2006.
4. Paul W. P. J. Grefen, Heiko Ludwig, Asit Dan, and Samuil Angelov. An analysis of web services support for dynamic business process outsourcing. *Information & Software Technology*, 48(11):1115–1134, 2006.
5. Jen-Yao Chung, Kwei-Jay Lin, and Richard G. Mathieu. Guest editors' introduction: Web services computing–advancing software interoperability. *Computer*, 36(10):35–37, 2003.
6. Cristian Mateos, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Adding semantic Web Services matching and discovery support to the Movilog platform. In *Proceedings of the IFIP 19th World Computer Congress (IFIP'06)*, volume 217 of *IFIP*, pages 51–60, 2006.

7. 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.
8. Ian Foster. Service-oriented science. *Science*, 308(5723):814–817, 2005.
9. Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Rec.*, 34(1):24–32, 2005.
10. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming*, in press, 2010.
11. M. Brian Blake and Michael F. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12(5):62–69, 2008.
12. John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis. Contemporary Web Service Discovery Mechanisms. *Journal of Web Engineering*, 5(3):265–290, 2006.
13. Rob McCool. Rethinking the Semantic Web, part II. *IEEE Internet Computing*, 10(1):96, 93–95, 2006.
14. Baoli Dong, Guoning Qi, Xinjian Gu, and Xiuting Wei. Web service-oriented manufacturing resource applications for networked product development. *Advanced Engineering Informatics*, 22(3):282 – 295, 2008. Collaborative Design and Manufacturing.
15. Jack Beaton Brad A. Myers Jeff Stylos Ralf Ehret Jan Karstens Arkin Efeoglu Sae Young Jeong, Yingyu Xie and Daniela K. Busse. *End-User Development*, chapter Improving Documentation for eSOA APIs through User Studies, pages 86–105. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009.
16. J. Pasley. Avoid XML schema wildcards for Web Service interfaces. *Internet Computing, IEEE*, 10(3):72–79, May-June 2006.
17. 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.
18. William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998.
19. 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.
20. 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.
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. Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
23. Sonja Zaplata and Winfried Lamersdorf. Towards mobile process as a service. In *Proceedings of 25th ACM Symposium On Applied Computing (SAC'10)*, pages 372–379, 3 2010.
24. 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.

25. Matjaz B. Juric, Ana Sasa, Bostjan Brumen, and Ivan Rozman. WSDL and UDDI extensions for version support in Web Services. *Journal of Systems and Software SI: Architectural Decisions and Rationale*, 82(8):1326–1343, 2009.