



**HAL**  
open science

# A Fully Automatic Approach to the Semantic Annotation of Web Service Descriptions

Cihan Aksoy, Vincent Labatut

► **To cite this version:**

Cihan Aksoy, Vincent Labatut. A Fully Automatic Approach to the Semantic Annotation of Web Service Descriptions. [Research Report] Galatasaray University, Computer Science Department. 2012. hal-01112241

**HAL Id: hal-01112241**

**<https://hal.science/hal-01112241v1>**

Submitted on 2 Feb 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# A Fully Automatic Approach to the Semantic Annotation of Web Service Descriptions

---

*Cihan Aksoy<sup>1,2</sup> & Vincent Labatut<sup>1</sup>*

*<sup>1</sup>Computer Science Department, Galatasaray University, Istanbul, Turkey*

*<sup>2</sup>TÜBİTAK, Istanbul, Turkey*

**Abstract:** The definition and use of a semantic layer in Web Services (WS) descriptions is a prerequisite to the automation of several important operations such as WS composition and discovering. For this reason, in the past years, many approaches have been proposed to either represent such high level information, or take advantage of it. In order to be properly tested and compared, these tools must be applied to an appropriate benchmark, taking the form of a collection of semantic WS descriptions. However, all of the existing publicly available collections are limited in terms of size or realism, leading to unreliable results. Large real-world syntactic (WSDL) collections exist, so an appropriate benchmark could be obtained through their semantic annotation. Due to the number of operations to process, performing this task manually would be costly in terms of time and efforts, though. A better solution would therefore be to use an automatic tool. The software mataws was precisely designed for this purpose. However, it suffers from certain limitations. In this work, we propose some modifications aiming at solving them. The resulting tool takes advantage of the latent semantics present not only in the parameter names, but also in the type names and structures. Concept-to-word association is performed by using Sigma, a mapping of WordNet to the SUMO ontology. After having described in details our annotation method, we apply it to the largest collection of real-world syntactic WS descriptions we could find, and assess its efficiency.

**Keywords:** Web Service, Semantic Web, Semantic Annotation, Ontology, WSDL, OWL-S, SUMO, WordNet.

## 1 Introduction

A Web service (WS) is a software component able to perform a set of well-defined tasks, and it can be remotely invoked through a stack of standard technologies [1]. The first WS appeared in 2001, when the core technologies WSDL and SOAP reached maturity. The former is used to describe the WS, and the latter for its invocation. More recently, RESTfull (Representational State Transfer) services emerged as an alternative WS technology. However, there is still ongoing work regarding how they should be formally described (e.g. WADL [2]), so in this work we focus on WS-\* services only.

WS are platform independent, so they can be invoked from any WS-compatible client. Moreover, WS technologies include standard ways of discovering previously unknown WS. These two properties theoretically allow one to automatically compose several *basic* WS, in order to get a so-called *composite* WS, and therefore provide a functionality of higher level. However, in practice this task requires a significant human intervention, both for the selection and combination of the WS. This is due to the way WSDL describes WS.

One can find three kinds of data in a WSDL file: first some low level properties regarding the network location of the WS and the communication protocols it supports; second some syntactic details indicating the names of the methods it provides, as well as the names and data types of their parameters; and third some semantic descriptions of the WS and its components. The first two explain *how* the WS can be used. They are compulsory and formal, so they can be used in an

automatic way. The third describes *what* the WS is doing. However, not only is it optional, but it is also expressed informally, as natural language text. It is clearly destined to humans, and this makes it hard for programs to take advantage of it. Yet, it is necessary for a program to access this information in order to achieve automatic discovery and composition [1, 3, 4].

The introduction of a formal semantic aspect in WS descriptions would allow to solve this problem, hence the development of specific languages, starting with DAML-S in 2002 [5]. This domain has since then expanded, and constitutes a very important field of WS research now, because the definition of a semantic description format is tightly related to the development of the technologies able to take advantage of it. In [1], the authors distinguish three axes: the semantic WS description itself, through the definition of a specific ontology; the framework allowing to take advantage of this description (how to publish, discover, compose, invoke such WS); and the WS architecture, in charge of providing the functionalities needed by the framework (how to reason on semantic descriptions, how to compare them, etc.).

Up to now, no industry standard has emerged regarding semantic WS. Several concurrent formats and technologies exist, leading to a profusion of research works. What interests us in this article, is the fact the ideas and tools resulting from these works must be tested. For this matter, one needs a large collection of semantic WS descriptions, in order to achieve statistical significance. Moreover, these descriptions must be realistic, so that the obtained results can be generalized to other WS.

As detailed in [6], only four significant collections of semantic descriptions exist. The first one is called *Dataset2* and is packaged with ASSAM, a WSDL annotator [7]. However, it contains only 164 descriptions. The SemWebCentral website [8] offers three interesting collections. *OWLS-TC4* is large enough, with 1083 descriptions, but its realism is subject to questioning because only a part of the files come from real-world WS [6]. The *SAWSDL-TC* collection is a subset of OWLS-TC based on a different semantic description language, so not only is it smaller, but it suffers from the same limitation. Finally, *SWS-TC* is a small collection of 241 files, on which very little information is available. These collections have been used as test beds in the literature [9, 10], but it is clear they do not comply with our criteria.

Alternatively, the desired benchmark collection could be obtained by annotating a set of WSDL files. The annotation of WSDL files is much different from other Web resources, due to the specific structure of the document. One of the difficulties is the lack of context, since the available information is made up of isolated words, and not full sentences. A few publicly available tools exist for this purpose, as described in [6]. *Radiant* [11] (from the Meteor project [12]) and *WSMO Studio* [13] are two Eclipse plugins designed to ease the manual semantic annotation of WS descriptions. So is the Web application described in [14]. All are completely manual, in the sense the user has to select the most appropriate concepts during the annotation process. Performing the annotation manually is at the same time difficult, costly, and prone to errors, so these tools do not seem adapted to our situation. *MWSAF* [12] and the previously mentioned ASSAM (Automated Semantic Service Annotation with Machine learning) [7] both take advantage of machine learning methods to propose the user a list of concepts he can choose from (after a learning phase). *SAWS* (Semantic Annotation of Web Services) [15] relies on a syntactic comparison of parameter and ontological concept names, in order to rank the concepts depending on their relevance. The tool presented in [16] adopts a similar approach, except it uses a semantic distance based on WordNet [17]. Thanks to this assistance, these tools are called *semi-automatic*. However, the core of the process remains manual, since the user still has to select the concept. Due to our constraint regarding the size of the collection, such an assisted approach does not seem appropriate.

More recently, fully automated approaches have been developed. The work in [18] is based on ontology alignment. First, WS are thematically grouped together, and associated to a domain ontology, i.e. a predefined ontology describing the concepts related to the identified theme (e.g. health, education, etc.). Second, for each description file, the service ontology is built by processing the parameter names. Then, concepts are associated to parameters by aligning (i.e. matching) the domain and service ontologies. Although promising, the implementation of this tool is not publicly available, and its authors do not specify which domain ontologies could be used, in practice. *Mataws*

(Multimodal Automatic Tool for the Annotation of WS), is another fully automatic tool, but this one is publicly available [6]. It adopts a so-called multimodal approach, in the sense it takes advantage of both parameter names and structured data types to associate ontological concepts to parameters, without human intervention. However, it suffers from some limitations. First, under certain circumstances it can output several concepts for one parameter, which is not supported by semantic WS description formats. Second, there is room for improvement concerning the quality of its annotations, in terms of relevance. Third, it is not able to annotate certain parameters.

In this article, we present an extended version of Mataws, aiming at solving these limitations. Our main contributions are the improvement of the existing components of the software, and the introduction of a new functional component which both dramatically raises the quality of the annotation and allows outputting a single concept. In section 2, we review the WS description-related points needed for a good understanding of the rest of the article. In section 3, we describe Mataws in details: we present the whole tool, including the parts already developed for the first version, but we highlight the modifications and additions adopted to obtain this second version. In section 4, we perform an empirical evaluation of our tool by applying it to a large collection of real-world WSDL files. We comment the obtained results, and compare them to those of the first version. Finally, we conclude with a discussion of the properties and limitations of our tool, and present some perspectives regarding how it can be extended to solve them.

## 2 Web Service Description

This section aims at giving the reader the prerequisites needed to understand the description of our tool in the next section. The first subsection is dedicated to WSDL, the current standard for WS description. It is not meant to be exhaustive: we focus only on the points relevant with our needs. The second subsection briefly explains what *semantic* WS descriptions are. Several competing languages exist, so we decided to stay general and not consider one of them in particular. The last subsection explains how we selected a large collection of realistic WSDL files, for two purposes. First, the manual analysis we conducted on these data allowed us to design our annotation algorithm. Second, the resulting software was applied on this collection for evaluation.

### 2.1 Syntactic Description & WSDL

The word *syntactic* is used here in opposition to the term *semantic*, and means this type of description focuses on the practical information needed to use the WS. The *de facto* standard is WSDL (WS Description Language), an XML-based language specified in 2001 [19] by the World Wide Web Consortium (W3C). It allows defining the public interface of WS, and more particularly: the communication protocol to be used, the format of the messages to be exchanged, the methods the client can invoke, and the location of the service. A second version was approved in 2007 to become a W3C recommendation. However it is much less used, so even if it contains significant changes, we chose to focus on version 1.1.

We are more particularly interested in the higher-level elements of WSDL, i.e. those related to the programmatic use of the service. From this perspective, WSDL files are very similar to RPC or RMI interfaces. They describe a list of *operations*, corresponding to the functions/methods the client can remotely invoke. The inputs and outputs of an operation are described under the form of two so-called *messages*: one for the inputs, the other for the outputs. A message is a group of *parts*, which correspond roughly to parameters, in terms of programming. Each one is characterized by a *name* and a *data type*. Figure 1 gives an example of simple WSDL file. It contains a single operation `myOperation` (highlighted in yellow) whose input and output messages are `myInMsg` and `myOutMsg`, respectively. The former (in red) contains two parts `myInParameter1` and `myInParameter2`, whereas the latter (in blue) has only one, named `myOutParam`.

Of course, all these elements are described in a neutral, implementation-independent way. For this matter, the data types are defined using the *XML Schema Definition* language (XSD), a 2001 W3C recommendation initially designed to let users define their own XML grammars [20]. Note WSDL

theoretically allows using other type definition languages than XSD, but as far as we know only this one has been used in production WS, up to now.

XSD types are generally separated in two groups: simple- and complex-content types. The former correspond to XML elements containing directly a value, and nothing else: no attribute, no other element. From the programming point of view, they can be implemented using the predefined simple types present in most languages: integer, float, string, etc., or derived types such as enumerations. The latter point out at elements containing at least one attribute and/or one element. Their implementation requires defining custom classes, for object-oriented languages, or types such as union, structures, arrays, etc. for non-object languages.

```

<?xml version="1.0"?>
<definitions name="MyService" ... >

<types>
  <complexType name="MySubType">
    <sequence>
      <element name="myField21" type="xsd:string"/>
      <element name="myField22" type="xsd:string"/>
    </sequence>
  </complexType>
  <complexType name="MyType">
    <sequence>
      <element name="myField1" type="xsd:integer"/>
      <element name="myField2" type="MySubType"/>
    </sequence>
  </complexType>
</types>

<message name="myInMsg">
  <part name="myInParam1" element="xsd:integer"/>
  <part name="myInParam2" element="xsd:string"/>
</message>
<message name="myOutMsg">
  <part name="myOutParam" element="MyType"/>
</message>

<portType name="SomePortType">
  <operation name="myOperation">
    <input message="myInMsg"/>
    <output message="myOutMsg"/>
  </operation>
</portType>

...

```

Figure 1. Excerpt of simple WSDL file (non-relevant parts have been removed).

XSD is powerful in the sense it allows defining many different sorts of types, especially for complex contents. However, between the constraints imposed by WSDL and the choices made by WS designers, only two kinds of data types turn out to be used in practice: simple-content types, and *sequence* complex-content types. For this reason, in the rest of this article we focus on these sorts of types. *Sequence types* lead to elements containing other elements of specific types and in a predefined order. They can be implemented as classes or structures, the internal elements corresponding to fields. The classes/structures can even be recursive, since each contained element can itself have a sequence type. In Figure 1, the input parameters `myInParam1` and `myInParam2` both have a predefined simple type: `integer` and `string`, respectively. The output parameter `myOutParam` has a custom type `MyType`, highlighted in orange. It is a sequence of two elements: an `integer`, and another element of type `MySubType` (in green) which contains itself two `string` values. In the rest of the article, when we mention complex-content types, we implicitly refer to those based on sequences.

## 2.2 Semantic Description

Semantic WS descriptions can be considered as the result of the application of the semantic Web to WS. The goal of the semantic Web is to formalize how the meaning of Web resources is represented, so that these can be efficiently processed automatically [21]. Concretely, this requires tagging resources, or parts of resources, with labels whose semantic is precisely known. Although

these tags can take various forms such as terms from a controlled vocabulary, ontological concepts are the preferred approach for WS. An ontology is a set of concepts, and the semantic relationships between them. Concepts are labels with a clearly defined meaning. The relational information encoded in the ontology allows building a hierarchical structure of concepts, which in turns can be used to perform automatic inference.

The annotation process, which consists in tagging the considered resource with the appropriate concepts, is completely dependent on both the nature of the resource and the automation goals. In other words, one will not annotate a text the same way one annotates a picture. And for a given kind of resource, the relevance of its various components might vary depending on the process one wants to automate. As shown in the previous section, WS descriptions are much different from other Web resources in terms of content. Moreover, their use also significantly differs: their semantic description aims at automating WS discovery, invocation, composition and monitoring [22]. Amongst the many elements constituting a WS description, these specific properties led to the identification of four thematic groups [23]: those concerning the inputs and outputs of the WS (*data semantics*), the process performed by the WS (*function semantics*), the constraints applied to the WS, such as the quality of service (*non-function semantics*) and the execution flow inside the WS (*execution semantics*).

Several languages were defined to represent semantic descriptions, the most prominent being OWL-S [22], WSMO [24], WSDL-S [25] and SAWSDL [26]. Although these allow annotating all parts of the description, the focus in the literature is on the data semantics, and more specifically the exchanged parameters. This is certainly due to the fact this information is particularly important to automate WS discovery and composition, two popular research fields. As mentioned before, our goal in this study is to provide the community a collection of WS descriptions usable as a benchmark, so we decided to focus on the data semantics, too. For the same reason, we decided to use OWL-S as the output language of Mataws, since it is the most widespread in the literature, and it is supported by the W3C. As we will explain later in further details, our software was designed to select an ontological concept for each input parameter. The form this concept takes in the outputted semantic description file is therefore of little importance, and no deeper knowledge of OWL-S is necessary to understand the rest of our explanations. Note also that this independence makes it possible to easily switch from OWL-S to another language without any modification other than the very last step of the process, consisting in generating the semantic description files.

### 2.3 Selection of a WSDL Collection

In order to design our annotation tool, we decided to adopt a data-driven approach. For this matter, it was necessary to first retrieve a collection of WSDL files which would be the most representative possible. We translated this representativeness constraint into two criteria: first the WSDL files need to represent real-world services in order to contain realistic data, and second the collection must be large so that it is general enough. An additional constraint is the use of English to describe WS, in particular when defining parameter names.

There are not a lot of WSDL collections publicly available, so it is possible to review them all here. The *SemWebCentral* website centralizes various resources related to the semantic Web [8]. It offers several collections of semantic WS descriptions using various semantic formats. Amongst them, the *OWLS-TC4* (OWL-S Service Retrieval Test Collection v.4) collection provides both WSDL and OWL-S descriptions for 1083 services. However, only a part of the descriptions concerns real-world WS, and those are not identified in the collection. Moreover, the way the rest of the WSDL files were obtained is not documented.

The *OPPOSum* (Online Portal for Semantic Services) website gathers various collections related to both semantic and syntactic WS descriptions [27] (including some of *SemWebCentral*). For our purposes, the most noticeable one is the *Jena Geography Dataset*, which contains 201 real-world descriptions from the domain of geography and geocoding. The size of this collection is therefore insufficient for our goal.

Several IEEE conferences included some challenges based on the processing of syntactic and/or semantic WS descriptions. This is the case of the 2005 IEEE International Conference on e-Business Engineering (ICEBE), and also of the joint IEEE Conference on Commerce and Enterprise Computing (CEC) and IEEE EE Conference on Enterprise Computing, E-Commerce and E-Services (EEE) between 2005 and 2010. During these competitions, the evaluation was performed using some benchmarks specially defined for the occasion, available on the websites of these conferences. These collections are very large: several thousands of WSDL files. However, they were all created artificially, which makes them inappropriate to our needs.

The *ASSAM project* (Automated Semantic Service Annotation with Machine learning) [7] includes a collection of WSDL files named *Full Dataset*. It gathers 816 real-world WS descriptions, containing 7877 operations in total. Those were gathered from salcentral [28] and XMethods [29], two web sites listing public resources. According to our investigations, it is the largest collection made up of real-world descriptions available up to now, which is why we conducted our analysis on these data.

### 3 Proposed Annotation Method

We manually studied the Full Dataset collection, in order to identify patterns consistent with our automatic semantic annotation objective. An iterative process consisting in introducing/removing various processing steps in/from our tool allowed us to determine the best workflow. In this section, we describe our tool in details. We do not give the various modifications tested during its development: but rather focus only on the final version instead. We first introduce its general structure, then its components, detailing their design and functioning. In this description, we highlight the differences between the first version of Mataws [6] and the one presented here.

#### 3.1 General Architecture

Mataws takes a collection of WSDL files as its input, and outputs the corresponding annotated descriptions under the form of OWL-S files. The focus is on the English language, which is expected to be used in the input files, and which is used in the output files. It was developed in Java, and uses various open source libraries, which are mentioned later. As shown in Figure 2, it is made up of three main parts. The first and last parts, *Input Component* and *Output Component*, are dedicated to the processing of the original WSDL files and the generation of the OWL-S files, respectively. The middle part is constituted of several *Core Components* (detailed in Figure 3) implementing the semantic annotation process itself.

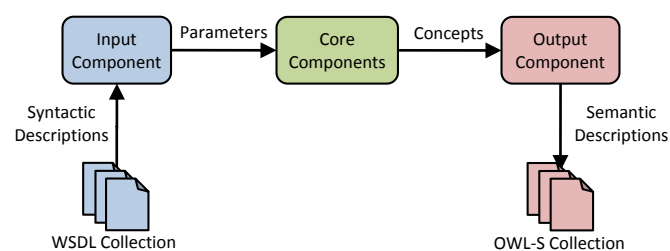


Figure 2. General Architecture of Mataws. The Core Components are detailed in Figure 3.

The *Input Component* is responsible for extracting the relevant information from the considered collection of WSDL files. Our annotation process focuses on parameters, which is why we need to retrieve their names, for all the operations defined in the collection. Moreover, in certain cases, we also need to know the name of their data types, and possibly their structure if the type has a complex content. This work is delegated to a parser previously developed by our research team [30], which produces a set of Java objects representing the extracted information. These objects are fetched to the Core Components, which will associate concepts to the parameters, therefore completing the objects. The *Output Component* is in charge for taking advantage of these completed objects in order to generate a collection of OWL-S files. It relies on an external library called Java OWL-S API [31],



which provides a programmatic read/write access to OWL-S files. Our tool can easily be extended to output other semantic WS description formats such as SAWSDL [26], WSDL-S [25], provided comparable APIs exist for them.

The organization of the Core Components is represented in Figure 3. There are four of them, each one in charge of a specific part of the annotation process. Each parameter is processed separately, through a recursive process. First, the *Preprocessor* receives the Java object representing the parameter, from the Input Component. Its role is to split some name into several words, which are then normalized and filtered. What we call a *name* here is a text string whose meaning is not always directly accessible, e.g. a concatenation of altered words. The Preprocessor is initially applied to the name of the received parameter. It produces a list of words, which is fetched to the next component: the *Selector*. This component is designed to output a single word able to represent the whole list. We call the resulting word the *representative word* for the considered name. Note it can be one of the words from the list sent by the Preprocessor, but it can also be a different, supposedly more meaningful one.

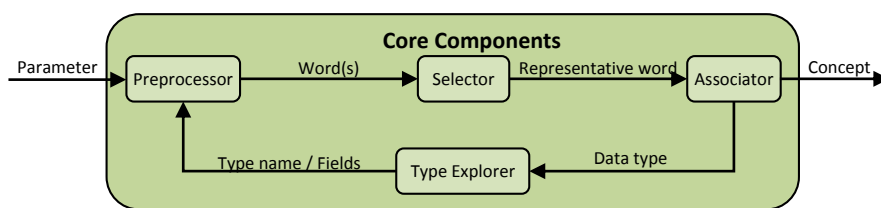


Figure 3. Core Components, corresponding to the central part of Figure 2.

The next step, performed by the *Associator*, consists in selecting an *ontological concept* able to formally represent the meaning of the representative word. If the *Associator* is successful and returns a concept for the representative word, this concept is linked to the considered parameter in the corresponding Java object. It can then be used later by the Output Component when generating the OWL-S file.

But, for various reasons explained later, it is also possible that the *Associator* cannot return a concept. In this case, the *Type Explorer* steps in and retrieves some properties related to the parameter data type. First, the analysis conducted on the parameter name can also be applied to the type name. Indeed, in WS it is common to define custom types, and their name can be sufficiently informative. Second, if the type name is not sufficient to select a concept, then one can consider its structure. For a complex-content type containing fields (i.e. the equivalent of a `struct` type in C programming), the same process can be applied again to each field. Those are sent to the *Preprocessor*, which can handle them like parameters since they have a name and a type. This results in a recursive process, whose end depends on the selection of a concept by the *Associator* (success) or the exhaustion of the data type (failure).

The Input and Output Components are not concerned with the business logic of our tool, so they do not require any further description. As a consequence, the following subsections are dedicated to the Core Components only. Note the general structure of the tool is the same for both versions of Mataws. However, in the second version, the *Preprocessor* and *Associator* were modified, and the *Selector* was added. The main differences are highlighted in the subsections dedicated to these components.

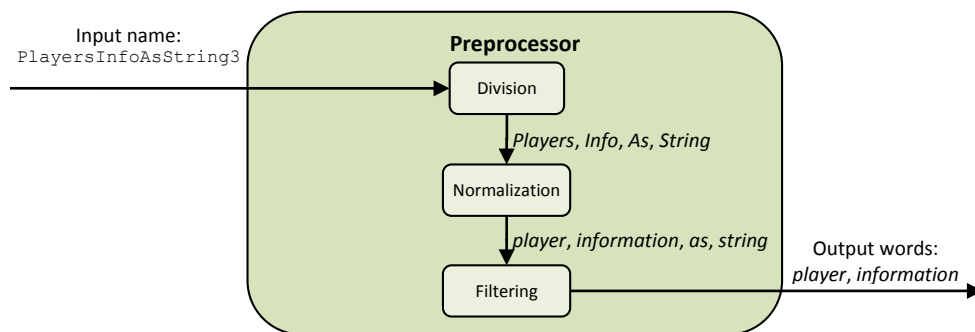
### 3.2 Preprocessor

The *Selector* and *Associator* can only process single words. However, the names defined in WSDL files do not usually comply with this requirement, for three main reasons. First, it is difficult to describe the meaning of a parameters, operations or types using a single word. As a result, names are most of the time made up of several concatenated words, separated by alternating lower and upper cases, or by using special characters such as underscores, hyphens, etc. Second, sometimes the resulting composite names are too long to be practical, so WS designers use abbreviations in



order to reduce their length. Finally, various forms of noise are also present, taking the form of meaningless digits, special characters and spelling errors.

Due to the number of possibilities, it is not possible to determine all forms of names one can find in a WS description. Instead of such an exhaustive approach, we adopted a data-driven method. We manually analyzed the collection of WSDL files selected in section 2.3, in order to identify properties and patterns regarding the parameter, type and field names. It became clear most of these names are built using classic programming conventions, such as those applied for the C or Java languages. It is therefore possible to define and apply a limited appropriate set of transformations to extract usable words from a name. Our name preprocessing consists of three steps: division splits the name in relevant substrings, normalization turns them into clean words, and filtering remove some irrelevant words. As an example, Figure 4 shows the preprocessing of the real-world parameter `PlayersInfoAsString3`, which will be commented in the rest of this section.



**Figure 4.** Algorithm of the Preprocessor, and application to an example: the parameter name `PlayersInfoAsString3`. The final result is a list of two words: *player* and *information*.

The *division* step is mainly responsible for breaking a name into several parts. For this matter, we used several templates presented in Table 1. Those were defined after the manual analysis mentioned earlier. They rely mainly on the presence of specific characters or typographic features in the considered names. Additionally, some cleaning is also performed during this step by removing non-letter characters. In the example from Figure 4, the Preprocessor takes advantage of the alternation of upper and lower cases to split the name `PlayersInfoAsString3` in 4 parts: `Players`, `Info`, `As` and `String`. The numeric end of the name is removed. Compared to the first version of Mataws, an important improvement was introduced, allowing to split words even in the absence of any typographical information. For instance, we can now break the name `username` down to the two words *user* and *name*. This feature relies on a Java library called *jWordSplitter* [32], which takes advantage of a dictionary to identify words in strings. The very last version focuses on the German language only, but we used an early version, able to process English too.

Template	Name	Result
Upper and lower case alternation	<code>PlayersInfo</code>	<i>Players, Info</i>
Case alternation with two first case upper	<code>AParameter</code>	<i>A, Parameter</i>
Underscore separation	<code>article_id</code>	<i>article, id</i>
Hyphen separation	<code>date-format</code>	<i>date, format</i>

**Table 1.** Split templates and examples.

The *normalization* step corresponds to three tasks, which are all required so that the Associator can efficiently work. The first consists in setting all letters to lowercase. As an example, consider Figure 4: all strings lose their initial capital during the normalization. The second task is to replace abbreviations, such as `usr`, with their full length equivalents, such as *user*. Here, a difficulty arises, because some strings can correspond to both a word and an abbreviation at the same time, and/or abbreviate different words. For example, `no` might be used to indicate the negation of the word following it in a concatenated name, such as in `no_limit` (i.e. absence of any limit). But it might also

be the abbreviation of the word *number* as in `OrderNo` (i.e. number of the order). The choice of the appropriate meaning is extremely dependent on the considered WS, and could require a human intervention. For this reason, we give the user the possibility to define his own list of abbreviations and corresponding words, as an external text file. We provide a general list of common abbreviations, which can be adapted to various domains of interest. In Figure 4, the abbreviation *info* is replaced by the full word *information*. The third and last task consists in replacing a word by its canonical form (i.e. lemma), an operation called stemming. For this matter, we take advantage of the *JAWS* library (Java API for WordNet Searching) [33], which gives a programmatic access to the well-known *WordNet* lexical database for the English language [17]. *WordNet* stores the inflectional forms of every word, so *JAWS* makes it possible to retrieve the canonical form associated to a given inflection. For instance, plural nouns or conjugated verbs are replaced by their singular and infinitive versions, respectively. In Figure 4, the string *players* (plural) is replaced by the word *player* (singular).

The role of the *filtering* step is to eliminate stop-words, i.e. words whose meaning is not relevant to our context. For instance, the word *parameter* is largely found in parameter names, but it does not bring any significant information, because we already know if the considered name refers to a parameter. For this reason, it can be regarded as noise and ignored. In Figure 4, the words *as* and *string* are also considered as stop-words. The result of the preprocessing of parameter `PlayersInfoAsString3` is therefore a list of two words: *player* and *information*. Like for abbreviations, stop-words are extremely context-dependent. For this reason, in *Mataws* their specification has been externalized, too. It takes the form of a text file, initially containing generic stop-words common to most WS. Of course, the user can modify it, in order to adapt it to his own situation.

### 3.3 Selector

The Selector is the most important difference with the first version of *Mataws* [6]. In the first version, this component did not exist: the Associator was applied directly to the set of words outputted by the Preprocessor, in order to identify an ontological concept for each one of them. It was therefore possible to get several concepts for a single parameter. But only one concept can be associated to a parameter in a description file. So an additional processing step was to be implemented, allowing the inference of a single concept. However, our first evaluation of *Mataws* showed the performances of the Associator did not seem as good as expected. It managed to identify a concept for most words, but not necessarily a relevant one [6]. Performing the previously mentioned inference on possibly inappropriate concepts could lead to flawed results. In order to tackle this problem, we decided to infer at an earlier stage, by considering directly the words coming from the Preprocessor. The Selector is in charge of this task, which results in the identification of a single word, called *representative word*. The Associator is then applied to retrieve the corresponding concept, which will be used when generating the semantic description file.

Like for the Preprocessor, we conducted a manual analysis on Full Dataset, the collection of real-world WS descriptions selected in section 2.3. This allowed us to identify various cases, which we believe are general enough to be relevant for most WS descriptions. In the rest of this subsection, we first present our analysis and its conclusions, and then describe the algorithm we derived from them.

#### 3.3.1 Data-Driven Analysis

The first goal of our manual analysis was to identify semantic relationships between the various words obtained for a parameter. We preprocessed Full Dataset and found occurrences of synonymy, hypernymy/hyponymy and holonymy/meronymy. We did not look at all for other relationships such as antonymy, because in our opinion, they would not be useful in our context. Two words are *synonyms* if they share a common meaning; this relationship is the only symmetric amongst those mentioned. One word is a *hypernym* of another if its meaning is more general. *Hyponymy* is the inverse relationship: one word is a hyponym of another one if its meaning is more specific. Hyper and hyponymy rely on a general-to-specific definition of inclusion, whereas holonymy and meronymy are whole-to-part based. One word is a *holonym* of another word if the thing it represents contains the

thing represented by the second word. The inverse relationship is *meronymy*: one word is a meronym for another one if the thing it represents is contained in the thing represented by the second word. Table 2 shows some general examples of such relationships for the word *bus*.

We noticed only a very few cases of synonymial relationships when analyzing Full Dataset, and those appeared to be accidental. Indeed, it seems very unlikely to find several times the same meaning in the description of a single parameter, because it would reflect a redundancy either in the parameter name or data structure. On the contrary, it is possible for the considered words to have several meaning, among which two happen to be similar, leading to a false positive when looking for synonyms. For instance, consider a name such as `coachBus` (meaning: the bus of the -sport- coach). An automatic search would find *coach* and *bus* to be synonyms, whereas in this situation they are not. In order to avoid incorrectly handling this case, which appears to be the most likely in this context, we decided the Selector would not look for synonymial relationships.

Relationship	Meaning	Examples
Synonymy	Both words have the same meaning	<i>autobus, coach, omnibus</i>
Hypernymy	First word is more general than the second one	<i>vehicle, transport</i>
Hyponymy	First word is more specific than the second one	<i>minibus, schoolbus</i>
Holonymy	First thing contains the second one	<i>fleet</i>
Meronymy	First thing contained in the second one	<i>roof, window, wheel</i>

Table 2. Examples of semantic relationships for the word *bus*.

The hyper/hyponymy relationship seems much more relevant. When such a direct relationship is detected between two words, we decided to retain the most specific one, because it has a more relevant meaning regarding the context. For example, the name `InterestPercentage` contains *interest* and *percentage*. As shown in Figure 5 the former can be considered as the hyponym of the later, so we can keep it as it is more specialized.

However, the relationship is not necessarily direct. Let us consider the parameter name `InterestRate`, for instance. As shown in Figure 5, on the one hand *interest* is a hyponym of *percentage*, which in turn is a hyponym of *proportion*. On the other hand, *proportion* is also a hypernym of *rate*. Even if *interest* and *rate* are not directly linked by one of the relationships from Table 2, they nevertheless share some sense since they have a common hypernym. In this case, one has the choice between three different words: the two original ones, and the common hypernym. It seems more appropriate to retain the latter, since his meaning is supposed to summarize both original words. It is important noticing there is a loss of meaning due to generalization, when replacing a word by a hypernym. In order to limit it, we decided after some experimentation to limit our search for a common hypernym to a distance of only two levels of the semantic tree.

Although looking for common hyponyms is also possible, this approach does not seem to be relevant. First, it is probable that, if two words have one common hyponym, then there are several others, which does not help us since we want to reduce the number of words associated to the parameter. Second, when two words have a common hyponym, it is likely there also exists a direct relationship between them, a case we already tested for. For instance, two synonyms obviously have common hyponyms.

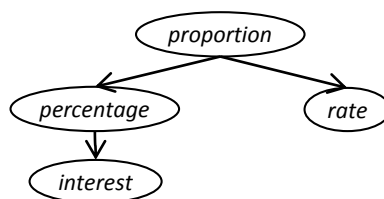


Figure 5. Example of semantic relationships between words. The source of an edge is a hypernym, the target is the hyponym.

We found two kinds of holo/meronymial relationships in Full Dataset: the direct ones concern lists of words derived from the same name, whereas indirect ones come from several field names, as found in complex-content types. In the first case, let us consider for example a name such as `CarWheel`: *car* is a direct holonym for *wheel*. Most of the time, we found other similarly formed parameter names in the same operation, such as `CarSeat` and `CarEngine`. We chose to keep the meronym, which is more specialized and therefore conveys the most precise meaning; moreover it is the most discriminant part of the name.

In the second case, we have a list of names coming from various fields related to the same parameter. This situation is explained in details in section 3.5, but to summarize: when no concept can be associated to a parameter name or type name, and if the type has a complex content, then its fields are taken into account. Each one is considered as a subparameter and is likely to represent an aspect of the main parameter. Provided a representative word can be selected for each field, they are often indirectly linked via a common holonym. Indeed, this kind of type is relevant to represent the various parts of some object of interest. For example, suppose the considered type contains some fields whose representative words are *handlebar*, *mudguard* and *pedal*. One of these words holonyms is *bicycle*, which is likely to be the representative word of the parameter.

But for most of the names, there is no obvious semantic relationship between the words extracted during the preprocessing. It is therefore necessary to propose another way of identifying a representative word. In the simplest case, the combination of the extracted words actually forms a compound word, or a well-known expression, for which we can find a single word synonym. For example, consider the parameter name `LastName`, which leads to the two words *last* and *name*: then *surname* constitutes a one word synonym for this compound. The expression *postalcode* (two words *postal* and *code*) can be summarized by the single word *postcode*.

In the case where the extracted words cannot be summarized, our analysis showed one of the extracted words is generally a noun whose meaning is central, and which is complemented by the other extracted words. It seems difficult to automatically retrieve the overall meaning of the expression. But it is worth noticing the central noun alone conveys the most important part of this meaning. Therefore, it constitutes a good representative word, even if part of the meaning is lost in the operation. In the English language, this central word is generally located at the end of the expression, which makes it easy to identify automatically. For instance, in the parameter names `BillingCountry`, `AuthorizationNumber`, `ApplicationName`, `AdvertTypeID` and `AdminEmail`, the central (and therefore representative) words are *country*, *number*, *name*, *id* and *email*, respectively. Note this case is very similar to the direct holo/meronymial relationships described earlier (cf. example `CarWheel`), except here we do not need to identify a direct semantic relationship. If there is no noun at all amongst the extracted words, then there is usually at least one verb. We consider it as the central word, since the rest are adjectives or adverbs, by definition. We consequently select it as the representative word. If several verbs are present, it is difficult to identify the most important, relatively to the considered context. The most objective approach is then to keep the verb the most frequently used in the English language.

### 3.3.2 Procedure Design

Based on our previous observations, we derived an algorithm aiming at obtaining the representative word for some list of words extracted during the preprocessing. It takes the form of a series of independent steps, represented in Figure 6. For each one of them, the goal is to reduce the number of words in the list, while minimizing the overall information loss. This can be realized either by suppressing some words considered irrelevant or neglectable, or by replacing several words by a new one, which supposedly summarizes them. In the latter case, it is necessary to start the whole process all over again, since the new word might trigger different conditions. This explains why the diagram presented in Figure 6 contains loops. As soon as only one word remains in the list, it is considered as the representative word and the process is over.

Our process relies largely on WordNet [17], accessed through the JAWS API [33], already mentioned for the stemming operation in the preprocessor. First, WordNet contains the necessary

semantic relationships described in Table 2. But we also use it to determine the grammatical nature of a word (verb, noun, etc.). Indeed, WordNet is able to provide all the grammatical roles associated to the various meanings of a given word. For instance, the word *clean* has 1 meaning for which it is considered as a *noun*, 10 meanings for which it is a *verb*, 18 for which it as an *adjective*, and 2 for which it is an *adverb*. For our purpose, we use an approximation by retaining the most frequent role of the word, based on the assumption it is the most likely to occur in our situation, too. Finally, the last advantage of WordNet is its compatibility with Sigma [34], the tool at the core of the Associator (described in section 3.4). As mentioned before, Sigma maps WordNet to SUMO [35], which means all the words obtained through JAWS can be associated to an ontological concept.

The different steps of the algorithm are ordered depending on how much of the original information they allow retaining: we check first the most favorable situations, and finish with the cases corresponding to the most approximate operations. Suppose we receive the following list of words as an input of the Selector: *customer, mailing, address, kitchen, lavatory, washroom, postal, code* (this example is purely artificial, it was defined for illustration purposes only). The 1<sup>st</sup> step is the stop condition, met when there is less than one word in the list. In the 2<sup>nd</sup> step, we use WordNet to check if certain words can be identified as a single expression. If it is the case, the concerned words are replaced by the expression in the list, and the process starts again. In our example, *postal* and *code* can be replaced by *postcode*, so the updated list is *customer, mailing, address, kitchen, lavatory, washroom, postcode*.

The 3<sup>rd</sup> step is concerned with direct hyper/hyponymial relationships. In our example, *mailing* and *address* correspond to this kind of relationship because *mailing* is a hypernym of *address*. As explained before, only the most specific word (the hyponym) is kept, so here we remove *mailing* and get the list *customer, address, kitchen, lavatory, washroom, postcode*. In the 4<sup>th</sup> step, we look for indirect hypernymial relationships. The word *bathroom* is a common hypernym for *lavatory* and *washroom*, so they are both replaced by *bathroom*, which summarizes them. The list thus becomes: *customer, address, kitchen, bathroom, postcode*.

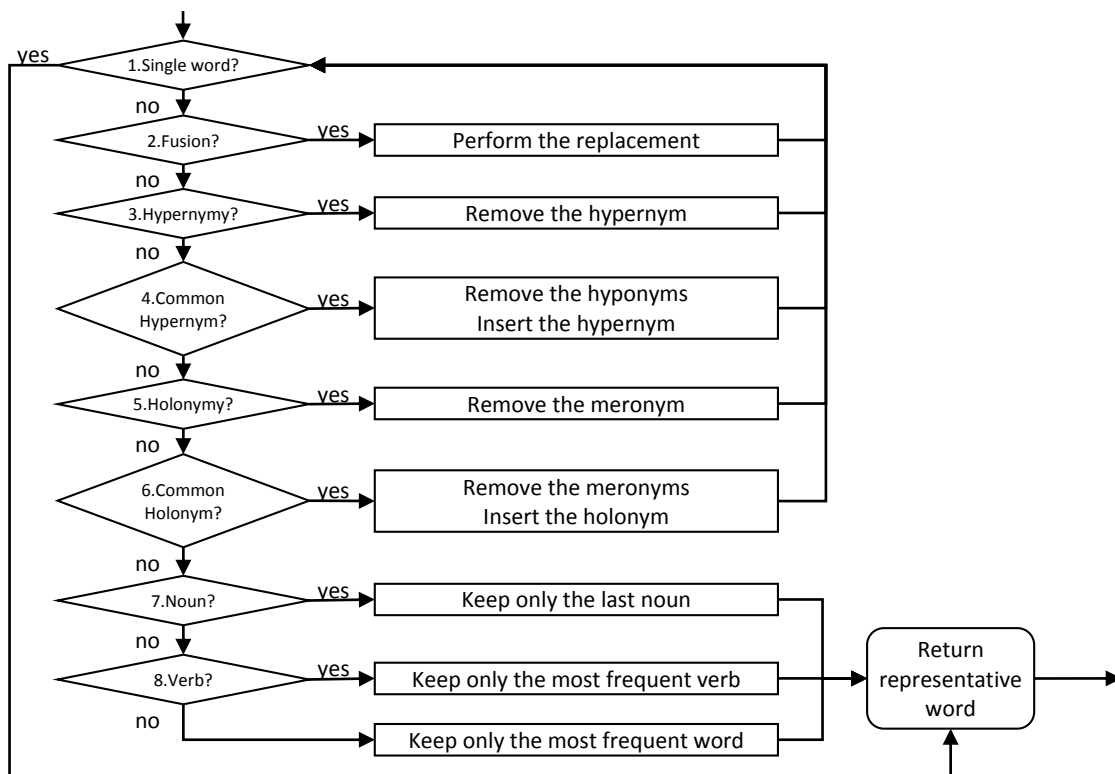


Figure 6. Algorithm of Selector with Examples.

The 5<sup>th</sup> step is dedicated to the detection of direct holo/meronymial relationships. In our list, *address* and *postcode* have this relationship, since *postcode* is a part of *address*, i.e. *postcode* is a meronym of *address*. In this case, we keep the meronym, i.e. *postcode*. The remaining words are therefore *customer*, *kitchen*, *bathroom*, *postcode*. The 6<sup>th</sup> step corresponds to the processing of words coming from different fields, i.e. we look for common holonyms. The words *kitchen* and *bathroom* correspond to parts of *home*, which is therefore a common holonym. In the list, we replace both words by *home*, obtaining: *customer*, *home*, *postcode*.

If the remaining words are not connected by any direct or indirect semantic relationships, we reach the last steps, which rely on grammatical information. The 7<sup>th</sup> one tries to detect nouns. In our case, we have three of them: *customer*, *home* and *postcode*: as explained earlier, we make the assumption the last ones are complements of the one placed just before, as in “the postcode of the customer’s home”. We therefore keep *postcode*. We only have a single word remaining in the list, so the process is complete and *postcode* is our representative word. We therefore do not reach the 8<sup>th</sup> step, which focuses on verbs.

If after all the previous steps, there are still several words in the list, then we chose the most frequent one (in the English language) amongst them.

### 3.4 Associator

Thanks to the process performed by the Selector, the Associator receives only a single word for each parameter. Its role is then to identify the most appropriate ontological concept to represent in a formal way the meaning of this word. For this purpose, we employ Sigma, which is a Java API implementing various ways of creating, testing, modifying and inferring on ontologies [34]. It is bundled with SUMO (Suggested Upper Merged Ontology) [35], the largest formal public ontology available up to now. It was first built using the SUO-KIF language [36], but it is now also available in OWL [37], a language compatible with OWL-S [22], the current output format of Mataws.

Word	SUMO Concept associated by Sigma
<i>buffalo</i>	HoofedMammal
<i>school</i>	EducationalProcess
<i>talk</i>	Communication

Table 3. Examples of concept associations.

Sigma is mainly designed for working on ontologies but it also implements a mapping between WordNet and SUMO, which is particularly interesting for us. Indeed, the words coming from the Selector have been outputted by JAWS, and are therefore contained in WordNet: this means Sigma should be able to find an ontological concept for each the word the Associator receives. Table 3 gives a few examples of the concept associations returned by Sigma.

It is important to note the mapping does not necessarily associate a concept of the same semantic level: it is often more general than the original word, like for example *buffalo* and *HoofedMammal* in table Table 3. So the Annotator output is not always very precise. However, compared to an automatic approach like ontology alignment [18] (cf. section 1), such a mapping has the advantage of having been defined and verified manually. It encodes the knowledge of experts, and is therefore more reliable than an algorithmic approach when it comes to identifying the ontological concept corresponding to a word.

Various methods give a programmatic access to the mapping, taking English words as input and returning SUMO concepts as output. In particular, for a given word, Sigma is able to return a list of concepts whose meaning can be expressed through the considered word. The reason of this multiplicity comes from the fact a word can have several distinct meanings. However, the way this list is ordered is arbitrary: it depends on the indexation order of the concepts in Sigma’s database, and not on any relevance criterion. In the first version of Mataws, the first concept in the list was always selected, for simplicity matters. But, as was noticed in [6], this was greatly decreasing the quality of the final annotation. In the second version, to improve this point, we take advantage of the

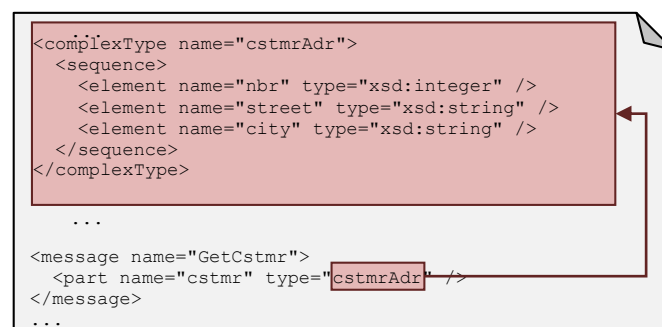


word usage frequency information available in WordNet. It allows us to identify the most frequent meaning of the word, and then select the concept corresponding to this specific meaning, amongst those returned by Sigma. The concept returned by the Associator is therefore more likely to correspond to the actual meaning of the word than in the first version of Mataws.

### 3.5 Type Explorer

Most of the time, the process implemented by the components described in the previous sections is sufficient to associate a concept to a parameter. However, there are some specific situations for which they do not succeed. First the Preprocessor might fail to break the name down to relevant words. This can be due, for instance, to an atypical way of forming the name, or even to errors in their definition, e.g. `myPaRameTr`. This is likely to prevent the Associator from identifying a concept. Second, if the parameter name is a stop word, or contains only noise and stop words, e.g. `AParameter_11`, then the Preprocessor will completely filter it. The Selector will therefore receive an empty list, and will not be able to identify a representative word to fetch the Associator. Third, Sigma very rarely fails to return a concept for certain words, even if they are correct English words.

Under one or several of these circumstances, the Associator is not able to return a concept for the considered parameter. Yet, the identification of such a concept is of course necessary to annotate the WS description. In order to overcome this problem, we take advantage of the latent semantics possibly conveyed by the data type of the parameter. It takes two different forms: first the name of the type, and second its complex content. This means our method cannot be applied to primitive types (`integer`, `string`, etc.). In this case, Mataws is definitely unable to identify a relevant concept for the considered parameter. This is signaled by associating a special symbol `NoMatch` representing the absence of concept in the generated OWL-S file. If the type is custom, it is likely to bear an explicit name, i.e. related to the information contained represented by the parameter. This name can be treated using the exact same process already applied to the parameter name.



```
<complexType name="cstmrAdr">
  <sequence>
    <element name="nbr" type="xsd:integer" />
    <element name="street" type="xsd:string" />
    <element name="city" type="xsd:string" />
  </sequence>
</complexType>

...

<message name="GetCstmr">
  <part name="cstmr" type="cstmrAdr" />
</message>

...
```

**Figure 7.** Excerpt from a real-world WSDL file, representing a parameter with complex-content XSD type.

But since the process is the same, it can fail for the same reasons described earlier. In this case, the content of the type can be analyzed, provided the type has a `sequence` complex content. As mentioned in section 2.1, such a type allows one element to contain a series of other elements. The parameter type is therefore very much like a structured type in C-like programming languages, i.e. it is made up of several fields. Note that for now we focus on `sequence` complex content because it is the most widespread, however the same approach can be extended to the other kinds of XSD complex content as well.

The fields have themselves a name and a data type, so our process can be applied on each of them separately. In the best case, if the field names are sufficient, this results in a list of representative words: one for each field. Those can then be combined by the Selector to get a single representative word for the whole parameter. If one (or more) name turns out to be insufficient, its type name can be used like we did before for the parameter. And if the type name is still not enough, then its (possibly) complex content can be ultimately exploited. This result in a recursive process: as



long as the element data type has a complex content, the process can be applied one step deeper, until the considered type has a simple content.

Figure 7 displays a part of a real-world WSDL file, in order to illustrate how complex-content types can be used. A parameter named `customer` has a type called `customerAdr`. This type has a complex content consisting of a sequence of one integer `number` and two strings `street` and `city`. The intended parameter and type names are `customer` and `customerAddress`, respectively. However, their meaning cannot be directly retrieved automatically, because the actual names are not explicit enough. Since a first pass is not sufficient to determine the associated concept, Mataws will consider the type content. The names of the elements in the sequence would lead to the words *number*, *street*, and *city*, which the Selector would have to combine, in order to obtain a single representative word for the parameter: *address*.

## 4 Experimental Validation

We tested the new Mataws version by applying it to the collection of real-world WS descriptions selected in section 2.3 (*Full Dataset* from the ASSAM project [7]). The first version of Mataws had been evaluated on the exact same collection, which gives us the possibility to study the effects of the modifications included in the second version.

The result of the automatic semantic annotation of WS descriptions can be evaluated in two different ways. We can obviously adopt a *quantitative* perspective, and consider the proportion of parameters for which Mataws was able to provide an ontological concept. But this type of evaluation is relatively limited, because even if one concept was associated to a parameter, this concept can be completely irrelevant. In the case of a real-world application, suppose the human user performs a second pass after Mataws, in order to manually annotate the parameters for which Mataws failed to identify a concept. Then, providing an incorrect concept is worse than providing none at all, because the user will not detect and correct these errors. It is therefore necessary to complete the quantitative evaluation with a *qualitative* one. This operation can be performed only manually, since it consists in trustworthily annotating the parameters and comparing the resulting concepts with those automatically selected by Mataws.

In this section, we first present the quantitative results of the second version and compare it with the first version. We then proceed similarly with the qualitative results, before discussing them.

### 4.1 Quantitative Evaluation

The evaluation of the first version of Mataws was constrained by the fact it possibly outputs several concepts for a single parameter, as described in section 3. For this reason, when evaluating it, we had considered a parameter was annotated if Mataws was able to return at least one concept. The new version outputs either a single concept or no concept at all, so there is no important change for the quantitative evaluation method, and the results can be compared directly.

The considered collection contains 9869 parameter instances in total, corresponding to 2465 unique parameters. We consider two instances correspond to the same parameter if they both have the same name and data type. In our previous work, we had only considered names, which explains why we found slightly more unique parameters here, compared to [6]. To perform the evaluation, we considered the rates of annotated parameters relatively to both views (instances and unique ones). The unique parameters rate represents how well the software performs when considering the heterogeneity of parameters. The parameter instances rate is more descriptive of the performance regardless of how parameters are distributed in the collection.

Table 4 displays both annotation rates for both versions of Mataws. The difference of 19.85 points observed for the second version between parameter instances and unique parameters rates is very similar to what was previously experienced for the first version (17.92 points). In both cases, this is due to the fact the parameters the tools could not annotate are amongst the most frequent in the considered WSDL collection.

Data	First Version		Second Version		
	Total	First Version	Second Version	Total	
All Parameter Instances	9869	7172	72.67%	7542	76.42%
Unique Parameters Only	2465	2233	90.59%	2373	96.27%

**Table 4.** Proportions of annotated parameters for both versions of Mataws.

The improvements applied to the second version allowed to significantly raise the score regarding both parameter instances (+3.75 points) and unique parameters (+5.68 points). Note all significance assessments in this section are performed using Student's  $t$ -test with  $\alpha = 0.05$ . We can conclude the second version is able to annotate a wider variety of parameters, and more efficiently.

The increase for unique parameters can be explained by the modifications introduced in the Preprocessor. Indeed, the parameters that the first version cannot annotate mostly have compound names such as `emailaddress` or `filedata`, or names taking a non-canonical form, such as `allocated` (conjugated verb) or `weeks` (plural noun). And as we mentioned in section 3.2, the second version can handle these kinds of names. However, a non-neglectable proportion of parameters remains non-annotated. These correspond mainly to names containing only stop-words, such as `Body` and `Return`, and associated with simple-content data types. Put differently, the parameters whose names and types do not convey sufficient information remain impossible to annotate. We call them *meaningless parameters*. The only way to solve this problem would be to consider other sources of information, such as the names of the `message` or `operation` elements containing the parameters (cf. section 2.1).

Studying the amount of annotated parameters allows us to evaluate the final output of Mataws. But it is also possible to consider words, which can be considered as partial results produced during the annotation process. Besides giving an insight of the internal behavior of our tool, this also provides a different way of comparing its two versions, since both of them internally handle words. Note that we considered the words fetched to the Associator, so there is a difference, though. For the first version, these words directly come from the Preprocessor, which means there can be several of them for a single parameter. For the second version, they come from the Selector, so we consider the representative word, which is unique and possibly the result of an additional process. This is illustrated by the fact that, for the second version, the number of word instances in Table 5 is the same than the number of parameter instances in Table 4 (which is the not the case for the first version).

Selected	First Version			Second Version		
	Total	Annotated		Total	Annotated	
All Word Instances	14437	12881	89.22%	9869	7542	76.42%
Unique Words Only	1185	816	68.86%	623	613	98.39%

**Table 5.** Proportions of annotated words for both versions of Mataws.

Table 5 displays the proportions of annotated words for both versions, including the case where no word at all could be fetched to the Associator. The first difference is directly linked to the previous remark: many more words reach the Associator in the first version than in the second, when considering word instances as well as unique words. Unlike for the parameters, the first version displays a decrease (−20.36 points) when comparing word instances to unique words. This means this version was good on frequent words.

On the contrary, the second version behaves like for the parameters, since its rate undergoes an increase of the same order than before (+21.97). The difference of behavior observed relatively to the first version indicates most of the word instances filtered by the newly introduced Selector were in fact annotable by the Associator. At first, this could be interpreted as a decrease of performance, compared to the first version. However, further analysis reveals 2307 out of these 9869 cases are completely filtered by the Preprocessor and Selector (i.e. the Associator receives an empty string). They correspond to meaningless parameters, such as `return`, `parameter` or `x`. Without them, the proportion of annotated word instances would be 99.73% (instead of 76.42%). Getting rid of such

parameters is actually a good thing, because even if the Associator is generally able to identify a concept for them, it is most of the time completely irrelevant: this results in a lowered annotation quality, as discussed in the next subsection.

When comparing the unique words rates for both versions, it appears the second one is largely more efficient (+29.53 points). This high score of 98.39% means almost all the words reaching the Associator in the second version are annotable, which confirms the importance of both the Selector and Preprocessor.

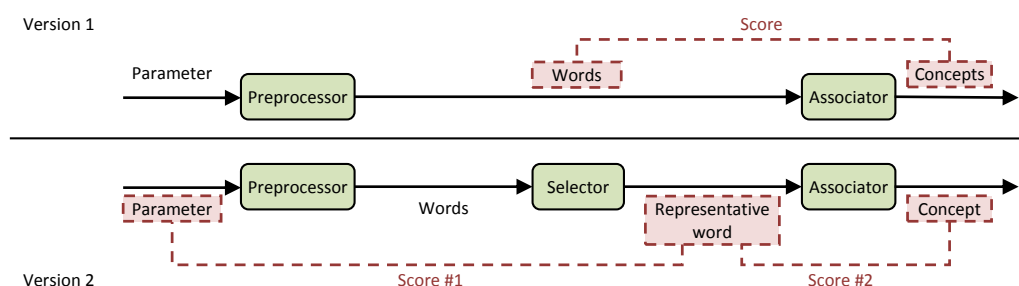
## 4.2 Qualitative Evaluation

The qualitative evaluation basically consists in grading the quality of the concepts outputted by Mataws, in terms of relevance with the original parameters. This operation must be conducted manually, since it is the only means we have to assess relevance.

Let us first review the method we used for the first version. The fact this version can output several concepts for a single input parameter made it more difficult to evaluate it qualitatively than quantitatively. For a parameter leading to several concepts, one solution could have been to evaluate only the most relevant one. But this would have introduced an important bias. So we decided to focus on an intermediary result instead of the input parameter: the words outputted by the Preprocessor. We associated a binary score to each pair of word-concept: 1 if Mataws was able to retrieve the correct concept for the considered word, 0 otherwise. The overall score was obtained by averaging all individual scores.

One of the important changes in the second version is the introduction of the Selector, which allows Mataws to output at most a single concept for each parameter. The constraint described above consequently does not apply anymore: it is possible to directly compare the final output of Mataws (concept) with the input parameter. However, considering the intermediary words, as we did for the first version, stays appealing for two reasons. First, it makes it possible to compare the qualitative results of the first and second versions, which could not be performed otherwise. Second, it allows a separated evaluation of the two semantic-related components of Mataws: the Selector and the Associator. This is doubly interesting, because the former was newly introduced, so we want to know how much improvement it brings; and the performances of the latter were questioned in our first work, which led to modifications we also want to assess.

In consequence, we finally applied the following two-stepped procedure. For each parameter, we first considered the representative word outputted by the Selector, and manually assigned a score expressing its relevance relatively to the parameter. We then similarly assigned a score to the concept outputted by the Associator, expressing its relevance to this representative word. For this matter, we took advantage of all available data: not only the parameter and data type names, but also contextual information such as the WS textual description or operation name. The first score allows us to evaluate how good the Preprocessor and Selector perform, whereas the second one stands for the performance of the Associator.



**Figure 8.** Comparison of the evaluation methods used in the two versions of Mataws. The Type Explorer is not represented for readability matters. In [6], the first version was evaluated by semantically comparing each word to its associated concept, in order to get a score. In the second version, it is possible to get two scores: the first one is obtained by comparing the initial parameter with its representative word, and the second one by comparing this word with the associated concept.

In order to consider a parameter has been successfully processed, we consequently need both scores to be high. Indeed, if the first is low, it means the representative word is not relevant, and thus even if the final concept is a good match for this word, it will fail to appropriately represent the original parameter. If the first score is high, but the second is low, then it means the tool failed to associate a relevant concept to an appropriate representative word, leading again to a incorrect result. The whole tool can consequently be evaluated by considering the minimum of both scores.

To be more precise and account for ambiguous cases, we chose to adopt a multivalued score ranging from 0 (not relevant at all) to 5 (completely relevant), instead of the binary values previously used. The score measures two different aspects of the annotation: correctness and precision. With the former, we aim at quantifying how much the associated meaning (representative word or concept) is semantically related to the considered object (parameter or representative word). The latter accounts for the fact the associated meaning sometimes subsumes the actual one. This is often the case for the concepts outputted by the Associator, due to the nature of the mapping implemented by Sigma (cf. section 3.4). Because of the new scale, the results from the first version were re-evaluated in order to get comparable scores. Figure 8 summarizes the evaluation procedures of both Mataws versions in a comprehensive way.

Let us first consider the performance at the level of the parameters, displayed in Table 6. As mentioned before, there are three different scores for the second version of Mataws: relevance of the representative word relatively to the original parameter (noted  $P$  vs.  $W$ ), relevance of the ontological concept relatively to the representative word ( $W$  vs.  $C$ ) and relevance of the ontological concept relatively to original parameter ( $P$  vs.  $C$ ). The latter is the minimum of the two former, and corresponds to the overall performance. All values displayed in Table 6 are average scores. Like before, we processed separately the results for parameter instances and for unique parameters. Unlike with the quantitative evaluation, and for all scores, the results are better for parameter instances than for unique parameters: +0.34, +0.16 and +0.36, respectively. This indicates the tool is good on the most frequent annotated parameters.

Data	Null Model		Second Version			
	Annotated	P vs. C	Annotated	P vs. W	W vs. C	P vs. C
Annotated Parameter Instances	7172	1.49	7542	4.18	3.45	3.04
Annotated Unique Parameters Only	2233	1.25	2373	3.84	3.29	2.68

**Table 6.** Average parameter annotation scores for the null model described in the text and the second version of Mataws.

The following remarks hold for both parameter instances and unique parameters. The best score is obtained for  $P$  vs.  $W$ , which means the representative words outputted by the Selector are semantically very close to the meaning of the corresponding parameters. The score decreases when considering  $W$  vs.  $C$ , which reflects the fact the Associator is not able to find a relevant concept for a part of the representative words. The overall score  $P$  vs.  $C$  is lower, meaning some of the representative words successfully annotated by the Associator were actually not relevant to the original parameter. The standard deviation for this last score is close to 2, so the somewhat central average (i.e. close to 2.5/5) hides the fact parameters are associated to either very relevant or very irrelevant concepts.

Comparing these results with those of the first version could not be done directly, since this version can output several concepts for a single parameter. Therefore, for the parameters in this situation, we decided to randomly select one concept as the final output. This amounts to defining a random Selector, which can be considered as a null model for the Selector defined in the second version. The obtained scores are displayed in the *Null Model* column of Table 6. Because of the different architecture of the first version, only the  $P$  vs.  $C$  score could be processed. The quality of the outputted concepts is more than doubled in the second version, for both instances and unique parameters. This shows some of the words extracted from the parameter in the first version have little relationship with its main meaning, and some additional process was therefore required to

perform a selection. It takes the form of the Selector in the second model, which leads to a significant improvement of the results.

We now focus on the assessment of intermediary words, shown in Table 7. Unlike for the parameter assessment, this time there is no need for a null model. Indeed, the first version of Mataws internally processes each word individually, before using Sigma to associate a concept. For both versions, and like for parameters, the average score increases when comparing words instances to unique word: +0.09 for the first version and +0.71 for the second. Thus, similarly to what was observed for parameters, the tools are better with frequently annotated words. There is a clear improvement of the annotation quality for the second version: 70% for word instances and 40% for unique words. This shows that the improvements we made not only in the Selector, but also in the Preprocessor, had a very positive effect on the annotation process.

Data	First Version		Second Version	
	Annotated	W vs. C	Annotated	W vs. C
Annotated Word Instances	12881	2.04	7542	3.45
Unique Annotated Words Only	816	1.95	613	2.74

Table 7. Average word annotation scores for both versions of Mataws.

Table 8 presents some more detailed results obtained with the second version, for the 15 most frequent parameters in the collection. The results obtained for the parameters ending in `-ID` (`ApplicID`, `UserID`, `AdminUserID`) demonstrate the interest of the improved Preprocessor and newly introduced Selector, leading to the representative word *identity*. The parameter `Result` illustrates the notion of meaningless parameter introduced in section 4.1. It is associated to the word *integer* and to the concept `Integer`. Its name is considered as a stop word in our context, and it has a simple data type. As explained before, the best Mataws can do in this situation is to take advantage of the type name.

Parameter	Freq.	Representative Word		Ontological Concept		
		Word	P vs. W	Name	W vs. C	P vs. C
<code>ApplicID</code>	228	<i>identity</i>	5	<code>TraitAttribute</code>	3	3
<code>Password</code>	223	<i>password</i>	5	<code>LinguisticExpression</code>	4	4
<code>UserID</code>	148	<i>identity</i>	5	<code>TraitAttribute</code>	3	3
<code>password</code>	114	<i>password</i>	5	<code>LinguisticExpression</code>	4	4
<code>username</code>	85	<i>user</i>	3	<code>experiencer</code>	4	3
<code>AdminUserID</code>	75	<i>identity</i>	5	<code>TraitAttribute</code>	3	3
<code>Result</code>	68	<i>integer</i>	4	<code>Integer</code>	5	4
<code>LicenseKey</code>	58	<i>key</i>	5	<code>Key</code>	5	5
<code>strGuidNotification</code>	58	<i>notification</i>	5	<code>RegulatoryProcess</code>	3	3
<code>UserName</code>	58	<i>user</i>	3	<code>experiencer</code>	4	3
<code>IsReleased</code>	52	<i>release</i>	3	<code>Demonstrating</code>	0	0
<code>accession</code>	46	<i>accession</i>	5	<code>Increasing</code>	0	0
<code>EmailAddress</code>	44	<i>address</i>	3	<code>uniqueIdentifier</code>	4	3
<code>MaxRecords</code>	40	<i>record</i>	3	<code>Text</code>	1	1
<code>date</code>	39	<i>date</i>	5	<code>Day</code>	4	4

Table 8. Qualitative results for the top 15 most frequent parameters: frequency (number of occurrences of the parameter), corresponding representative word (the word itself and its associated score), ontological concept (concept and score) and overall score.

The differences observed between the representative word and concept scores justify the necessity to distinguish the Selector and the Associator in terms of performance. The former extract a relevant representative word for most of the listed parameters: *identity* for `ApplicID`, *notification* for `strGuidNotification`, *key* for `LicenceKey`, etc. It is not as clear for the latter: `TraitAttribut` for *identity*, `experiencer` for *user* and `Key` for *key* seem fairly relevant, but `LinguisticExpression` and `RegulatoryProcess` for *password* and *notification*, respectively,

seem too general to be close enough to the actual meanings. This is due to the fact the mapping implemented by Sigma sometimes associates an ontological concept which is not directly equivalent to the considered word, semantically, but rather subsumes it, as explained in section 3.4. Some concepts are completely irrelevant, e.g. *Increasing* for *accession* and *Demonstrating* for *release*. These problems could be solved by replacing Sigma by a similar but more precise tool. However, such software, which would implement an equivalent mapping from the English language to an ontology, does not exist to the best of our knowledge. It seems more realistic to keep Sigma as the base of our Associator, and refine its results through some additional processing based on complementary NLP tools.

## 5 Conclusion

In this paper, we presented an improved version of our tool Mataws, which can semantically annotate syntactic WS descriptions in a fully automatic manner. As in the first version [6], the process is based on the mapping of the WordNet [17] lexicon to the SUMO ontology [35] implemented in Sigma [34], and on a multimodal approach consisting in taking advantage not only of the parameter names, but also of their data type names and structures. In the second version, we improved two components of Mataws and introduced a new one. First, we modified the Preprocessor so that it can handle compound and non-canonical words. Second, we tuned the Associator in order to select a more relevant concept when several ones are possible for a given parameter. Third, we defined the Selector, whose role is to identify the most relevant word, relatively to a given parameter, amongst the list outputted by the Preprocessor.

We applied Mataws to ASSAM Full Dataset [7], the largest available collection of real-world WSDL files, and evaluated the resulting collection of OWL-S files. The proportion of annotated parameters exceeds 75% of the collection. Further analysis of the results shows the non-annotated parameters would be hard to process even to humans, due to the lack of context (e.g. parameters simply called *parameter*). This improved filtering is mainly due to the modifications undergone by the Preprocessor. When considering the quality of the annotation, i.e. the relevance of the concepts associated to the parameters, the second version is significantly better, with an average grade exceeding 3/5. This is due in part to the improvement of the Associator, but most of all to the new Selector component: the analysis of partial results extracted during the process shows the semantic process it implements has a direct effect on the relevance of the outputted concepts.

The first contribution of our work is the improved Mataws tool. The second contribution is the result of the annotation process: a collection of semantic WS descriptions, which can be used as a benchmark to test some of the many methods developed specifically to take advantage of this semantic aspect. Moreover, note the tool is open source, relies on freely available libraries, and is freely available itself<sup>1</sup>. The files in the produced semantic collection follows the OWL-S format [22], a W3C-supported technology, and the collection is itself publicly available. It is bundled with a table containing the detail of the manual annotation work we performed when evaluating our tool.

As we mentioned earlier, it seems difficult to increase the performance of Mataws relatively to the proportion of annotated parameters. However, there is room for improvement regarding the quality of the annotation. Our experimental evaluation showed the main problem comes from the Associator, which does not always pick the most relevant concept when a word can be associated to several ones. We see two possible, non-mutually exclusive solutions to this problem: first, take advantage of some additional information to improve the concept selection; and second, use a different method to retrieve the concept. Mataws does not use certain parts of the WSDL files, such as the names of messages or operations, and the optional natural language descriptions (cf. section 2.1). Those could be exploited directly, like we did for parameter names and types. Another possibility is to adopt the approach seen in [18], in order to identify the domain ontology of the WS. This would allow the Associator to work on a subset of concepts, and therefore decrease the risk of

---

<sup>1</sup> <http://code.google.com/p/mataws/>



selecting an irrelevant one. Regarding the second solution, potential alternatives to Sigma exist, although their use is generally less direct, such as DBpedia [38] or Wikipedia-based dictionaries like the one described in [39]. Comparing and combining the results outputted by several such tools constitutes a promising perspective.

## 6 References

1. Cabral, L., Domingue, J., Motta, E., Payne, T., Hakimpour, F.: Approaches to Semantic Web Services: An Overview and Comparisons. LNCS 3053, 225-239 (2004)
2. Hadley, M.: Web Application Description Language, <http://www.w3.org/Submission/wadl/>
3. Stollberg, M., Feier, C., Roman, D., Fensel, D., Ide, N., Cristea, D., Tufi, D.: Semantic Web Services - Concepts and Technology. Language Technology, Ontologies, and the Semantic Web. Kluwer Publishers (2006)
4. McIlraith, S., Cao Son, T., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems 46-53 (2001)
5. Paolucci, M., Sycara, K.: Autonomous Semantic Web Services. IEEE Internet Computing 7, 34-41 (2003)
6. Aksoy, C., Labatut, V., Cherifi, C., Santucci, J.-F.: Mataws: A Multimodal Approach for Automatic Ws Semantic Annotation. Communications in Computer and Information Science 136, 319-333 (2011)
7. Hess, A., Johnston, E., Kushmerick, N.: Assam: A Tool for Semi-Automatically Annotating Semantic Web Services. International Semantic Web Conference (2004)
8. InfoEther, BBN Technologies: Semwebcentral.Org Website, <http://www.projects.semwebcentral.org/>
9. Ma, J., Zhang, Y., He, J.: Web Services Discovery Based on Latent Semantic Approach. In: ICWS, (2008)
10. Skoutas, D., Sacharidis, D., Kantere, V., Sellis, T.: Efficient Semantic Web Service Discovery in Centralized and P2p Environments. Lecture Notes in Computer Science 5318, 583-598 (2008)
11. Gomadam, K., Verma, K., Brewer, D., Sheth, A.P., Miller, J.A.: Radiant: A Tool for Semantic Annotation of Web Services. 4th International Semantic Web Conference (2005)
12. Patil, A.A., Oundhakar, S.A., Sheth, A.P., Verma, K.: Meteor-S Web Service Annotation Framework. 13th international conference on the World Wide Web (2004)
13. Dimitrov, M., Simov, A., Momtchev, V., Konstantinov, M.: Wsmo Studio - a Semantic Web Services Modelling Environment for Wsmo (System Description). Lecture Notes in Computer Science 4519, 749-758 (2007)
14. Budinoski, K., Jovanovik, M., Stojanov, R., Trajanov, D.: An Application for Semantic Annotation of Web Services. 7th International Conference for Informatics and Information Technology (2010)
15. Salomie, I., Chifu, V.R., Giurgiu, I., Cuibus, M.: Saws: A Tool for Semantic Annotation of Web Services. In: IEEE International Conference on Automation, Quality and Testing, Robotics, 387-392, Cluj-Napoca, RO (2008)
16. Bouchiha, D.: Semantic Annotation of Web Services. 4th International Conference on Web and Information Technologies (2012) 60-69
17. Miller, A.G., Fellbaum, C., Tengi, R., Langone, H., Ernst, A., Jose, L.: Wordnet, <http://wordnet.princeton.edu/>
18. Canturk, D., Senkul, P.: Semantic Annotation of Web Services with Lexicon-Based Alignment. IEEE World Congress on Services (2011) 355-362
19. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1, <http://www.w3.org/TR/wsdl>
20. Fallside, D.C.: Xml Schema Part 0: Primer, <http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/>
21. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001)
22. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: Owl-S: Semantic Markup for Web Services <http://www.w3.org/Submission/OWL-S/>
23. Sheth, A.P.: Semantic Web Process Lifecycle: Role of Semantics in Annotation, Discovery, Composition and Orchestration. In: Workshop on E-Services and the Semantic Web, (2003)
24. de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., König-Ries, B., Kopecky, J., Lara, R., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., Stollberg, M.: Web Service Modeling Ontology (Wsmo), <http://www.w3.org/Submission/WSMO/>
25. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K.: Web Service Semantics - Wsdl-S, <http://www.w3.org/Submission/WSDL-S/>
26. Farrell, J., Lausen, H.: Semantic Annotations for Wsdl and Xml Schema, <http://www.w3.org/TR/sawSDL/>
27. Küster, U., König-Ries, B., Krug, A.: Opossum - an Online Portal to Collect and Share SWS Descriptions. (2008) 480-481
28. Salcentral: Salcentral.Com Website, <http://www.salcentral.com>
29. XMethods: Xmethods.Net Website, <http://www.xmethods.net/ve2/index.po>
30. Cherifi, C., Rivierre, Y., Santucci, J.-F.: Ws-Next, a Web Services Network Extractor Toolkit. In: 5th International Conference on Information Technology, (2011)
31. Sirin, E., Parsia, B.: The Owl-S Java Api. In: 3rd International Semantic Web Conference, (2004)
32. Naber, D.: Jwordsplitter, [http://www.danielnaber.de/jwordsplitter/index\\_en.html](http://www.danielnaber.de/jwordsplitter/index_en.html)
33. Spell, B.: Java Api for Wordnet Searching, <http://lyle.smu.edu/~tspell/jaws/>
34. Pease, A.: Sigma Knowledge Engineering Environment, <http://sigmakee.sourceforge.net/>
35. Niles, I., Pease, A.: Towards a Standard Upper Ontology. In: International Conference on Formal Ontology in Information Systems, Ogunquit, US-ME (2001)
36. IEEE: Suo-Kif (Standard Upper Ontology Knowledge Interchange Format), <http://suo.ieee.org/SUO/KIF/suo-kif.html>
37. McGuinness, D.L., Harmelen, F.: Owl Web Ontology Language, <http://www.w3.org/TR/owl-features/>
38. Universität Leipzig, Freie Universität Berlin, OpenLink: Dbpedia.Org Website, <http://wiki.dbpedia.org>
39. Spitzkovsky, V.I., Chang, A.X.: A Cross-Lingual Dictionary for English Wikipedia Concepts. In: 8th International Conference on Language Resources and Evaluation, Istanbul, TR (2012)