SEVENTH FRAMEWORK PROGRAMME
THEME 3
Information and communication Technologies

PANACEA Project

Grant Agreement no.: 248064

Platform for **A**utomatic, **N**ormalized **A**nnotation and
Cost-**E**ffective **A**cquisition
of Language Resources for Human Language Technologies

D6.4
Lexical Merger

Dissemination Level:	Public
Delivery Date:	05/11/12
Status – Version:	Final
Author(s) and Affiliation:	Riccardo Del Gratta, Monica Monachini (CNR-ILC), Maurizio Tesconi, Matteo Abrate, Angelica Lo Duca (CNR-IIT), Laura Rimell (UCAM), Núria Bel, Muntsa Padró (UPF).

Related PANACEA Deliverables:

D6.1	Technologies and Tools for Lexical Acquisition
D6.3	Monolingual Lexicon for Spanish, Italian and Greek of 100.000 words for a particular domain
D7.4	Third evaluation report

Table of Contents

1 Introduction

This document describes the experiments on the merging of lexical resources performed during the project and the development of two merging components for LMF lexicons.

The challenge of lexical merger in PANACEA was two-fold. One goal is to be able to integrate the lexical resources produced by the PANACEA components into a single multi-level lexicon, the second is to merge acquired lexical resources into an existing one. For the latter goal, we focused on the merging of syntactic (Subcategorization), for the latter we focused on the integration of the level of linguistic descriptions of resources produced by the components integrated in the platform, i.e. SCF, MWE, and Lexical Classes lexicons.

The work on merging performed within the project was driven by the attempt to experiment with and devise different merging methodologies that could be as general as possible in order to reduce manual intervention.

As the LMF ISO standard was chosen as the Traveling Object (TO) for lexicons, the multi-level merger was designed to deal with LMF and its structure and format is tied to the TO (which specifies the format for Subcategorization Frames -SCFs-, Multiword -MWEs- and lexical Classes, -LCs-)1.

2 Development of Tools and Technologies: Merging of dictionaries

2.1 Merging of Subcategorization Frames

Within this task, three main approaches have been experimented with. Two of them resulted in services integrated into the platform.

2.1.1 Merging for increased precision

UCAM work focused on the experimentation of a new method for merging of Subcategorization information automatically acquired using two different parsers with the goal of acquiring a higher precision SCF resource, i.e. where only the information that the two resources agree on is retained.

Differently from previous works, e.g. (Crouch and King, 2005; Molinero et al., 2009), here the focus is on merging the intersection between two resources. Treating language resource merging as (roughly) a union operation seems appropriate for manually developed resources, or in general when coverage is a priority. However, when working with automatically acquired resources, it may be worthwhile to adopt the approach of merger by intersection.

UCAM tried to reduce the noise that the taggers and parsers add to the automatic SCF acquisition, by combining two lexicons built with different parsers.

For the experiment – performed on English data – the parsers used are the RASP parser and the unlexicalized Stanford parser. SCF are acquired from both outputs by an adapted version of the SCF acquisition system of (Preiss et al., 2007), which is a rule-based classifier that matches the Grammatical Relations (GRs) for each verb instance with a corresponding SCF. Since the classifier is based on the GR scheme adopted for RASP and the scheme of the Stanford parser is different, a new version of the classifier has been developed for the

[1] Note that SPs are not included in the TO at this time; since state-of-the-art SP models specify probabilistic relations between any verb-argument pair, they are not easily represented in a flat lexicon format. As anticipated in D6.1, inclusion of SPs is left for future work.

Stanford output. In fact, Despite commonalities between the GR scheme of (Briscoe et al., 2006) and the SD scheme, the realization of a particular SCF exhibits a number of differences across schemes.

Next a parser combination step is performed, creating a new set of classified verb instances by retaining only instances for which the two classifiers agreed on the SCF. Here the classifier output is merged on a sentence-by-sentence basis and not on a verb-by-verb basis to reduce errors of both parsers to pass through the pipeline.

Finally, a lexicon building step has the task to amalgamate the SCFs hypothesized by the classifier for each verb lemma. SCFs left underspecified by the classifier are also treated at this stage.

2.1.1.1 Gold standard

For testing and evaluation purposed, the gold standard of (Korhonen et al., 2006b) has been used, which consists of SCFs and relative frequencies for 183 general-language verbs, based on approximately 250 manually annotated sentences per verb. The verbs were selected randomly, subject to the restriction that they take multiple SCFs. The gold standard includes 116 SCFs.

Details of the method and the experiment are given in Rimell, Poibeau and Korhonen (2012).

2.1.2 Generic/Customisable mapping based on LMF

CNR-ILC has developed a merger of LMF lexicons and performed experiments on the merging of Subcategorization frame information, as described in Del Gratta et al. (2012). ILC decided to use the Lexical Mark-up Framework (Francopoulo et al., 2008) since it defines an abstract meta-model for the construction/description of computational lexicons. LMF is organized in several different packages: each package is enriched by specific data categories used to adorn both the core model and its extensions.

Data categories can be seen as the linguistic descriptors that are used to instantiate each entry and play a crucial role in the merging of two SCFs lexicons.

This merger focuses on the LMF syntax extension where the Syntactic Behaviour represents the basic building block and encodes one of the possible behaviours of a Lexical Entry. A detailed description of the syntactic behaviour of a lexical entry is further defined by the Subcategorization Frame object, which is the "heart" of the syntax module.

SubCategorization Frame is used to represent one syntactic configuration, in terms of Syntactic Arguments and may be shared by different lexical entries. In other words SubCategorization Frames are verb independent and can be linked by Syntactic Behaviours of different verbs sharing the same argument structure. In addition to the argument structure represented by the SCF, the Syntactic Behaviour (SB) also encodes other syntactic properties of the entry: the auxiliary.

Figure 1 below shows the components of the LMF modelled to merge two SCFs lexicons.

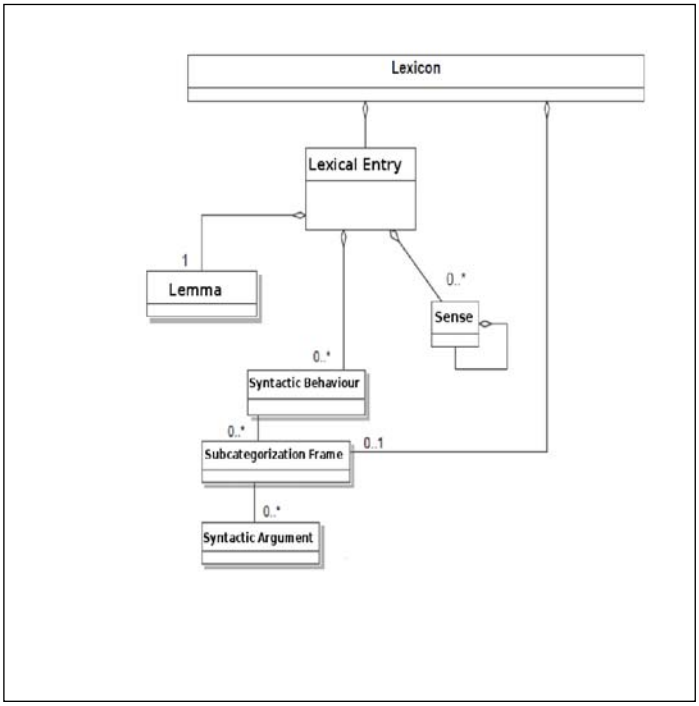

Figure 1: LMF core and syntax extension

Moving from model to software, a generic lexicon (L) can be modelled as a combination of lexical objects and their relations:

L = Lmorpho + Lsyn + Lsem +.... = Lmorpho + Link(Lsyn, Lmorpho) + Lsem +...

where the role of the link between the syntactic and morphologic extensions is played by the Syntactic Behaviour.

2.1.2.1 Software

ILC built a prototype which presents the architecture described in Figure 2.

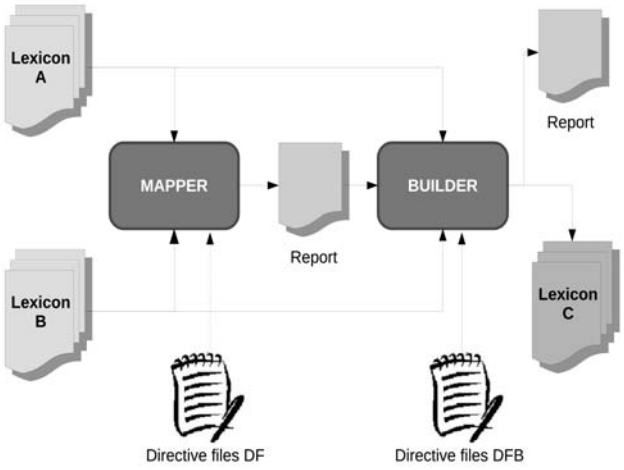

Figure 2: Basic architecture

This prototype takes two LMF lexicons, A and B, and a set of directives in input and outputs one lexicon which contains common information between the two lexicons.

The prototype consists in two main modules:

- Mapper: The Mapper component takes two lexicons (A and B) and a set of Directive Files; the Mapper produces a report document which lists all entries of lexicon A that have a potential match with entries of lexicon B according to given directives. The report is potentially meant both as output for the end-user and as input for the Builder component;
- Builder: The Builder component merges objects, SCFs and SBs, from the two lexicons. It takes the report both lexicon (A and B), and a set of directive files in input; it is responsible for producing the merged lexicon;
- Mapper and Builder Directives: these directives are used at different levels. They define a sort of (basic) files define semantic interoperability for specific features, for example that what is called *realization* in lexicon A is called *syntacticConstituent* in lexicon B;
- A rule to apply for defining equivalence among syntactic arguments (SAs). The software checks if some features, for example function, introducer, realization (but also others) are the same in the syntactic arguments structure in both lexicons. If two SCFs share the same SAs structure are defined as equivalent;
- A rule for managing auxiliaries in SBs: SBs which point to equivalent SCFs are defined equivalent if they have the same auxiliary;
- Thresholds for Cosine Similarity. The software is able to calculate the similarity between two SCFs from their SAs structure: the more two SAs have common features, the more two SCFs are similar.
- Directives make Mapper and Builder very flexible. They allow users to set the "constraints" under which entries from different lexicons can be "merged". The constraints can be more or less strict on the basis of the desired outcome.

2.1.3 Merger at work

As previously exposed, the software reads some directives before taking some decisions. Directives are simple files which can be totally defined by the users which can define their scenarios. In this chapter the format and the values of the directive files are described.

2.1.3.1 Directive feature_mapper

This directive defines which features of the syntactic arguments must be checked. It plays two distinct roles: the first is to list such features; the second is to map the same features between the two lexicons. The format is value_lexicon_A (tab) value_lexicon_B and it is coherent with the two input lexicons, A and B:

format lexA (tab) lexB

Lexicon A	**Lexicon B**
Function	function
syntacticConstituent	realization
Introducer	introducer

Table 1: mapping among features

This directives maps the features of the SAs in lexicon A on those of lexicon B. The software uses this directive to:

- count the features in SAs structure
- substitute the values in B with A's

The directive is used to establish equivalence among syntactic arguments such the following:

Lexicon A	**Lexicon B**	**Equivalent**
<Syn.... > <feat att="function" val="subj"/> <feat att="**syntacticCostituent**" val="np"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization** " val="np"/> <feat att="introducer" val="to"/> </Syn....>	YES. Both arguments have three features with the same attributes and values. Please note the **bold** which reflects the mapping.
<Syn.... > <feat att="function" val="subj"/> <feat att="**syntacticCostituent**" val="np"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization** " val="np"/> </Syn....>	NO. Different number of SAs
<Syn.... > <feat att="function" val="subj"/> <feat att="**syntacticCostituent**" val="***np***"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization** " val="***pp***"/> <feat att="introducer" val="to"/> </Syn....>	NO. Both arguments have three features with the same attributes but different values. Please note the **bold** which reflects the mapping but also the ***bold italic*** which shows different values.

Table 2: Equivalence between two syntactic arguments according to features_mapper directive

2.1.3.2 Directive mapper.properties

This directive contains instructions for both the mapper and builder components as well as for the report definition. Essentially, those instructions tell the mapper how many features two SAs must share to be defined equivalent and which is the strategy to consider equivalent two features; they tell the builder which attribute of the Syntactic Behaviour should be considered in addition to the equivalence of the SCF they point to; tell the report to include those SCFs which are in a predefined interval of similarity. The format is key=value (tab) comment

#key=value (tab) comment

Mapper	Comment

# threshold 	
threshold_min = 2
threshold_max = 3
	SAs must share at least two features to be considered equivalent

# features_equal_by = "same_att_and_val" or "same_att" 	
features_equal_by = same_att_and_val
	Features are equivalent if they have the either the same atribute or the same combination attribute+value

Table 3: Mapper specific directives

Builder	Comment

# auxiliaries 	
# true || false 	
check_auxiliaries = true
	SBs which point to equivalent SCFs must have the same auxiliary

# features_equal_by = "same_att_and_val" or "same_att" 	
features_equal_by = same_att_and_val
	Features are equivalent if they have the either the same attribute or the same combination attribute + value

Table 4: Builder specific directives

Report	Comment

# similarity 	
sim_min = 0.5
sim_max = 1
	The report contains SCFs whose similarity lies between the specified interval.
	The report is read by users which can manually change the similarity among SCFs.

Table 5: Report specific directives

Using these directives the results reported in table 2 can change accordingly:

Lexicon A	Lexicon B	Equivalent
….	….	…..
<Syn.... > <feat att="function" val="subj"/> <feat att="**syntacticCostituent**" val="np"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization** " val="np"/> </Syn....>	**YES**. Different number of SAs but two features are equivalent, so the threshold_min = 2 parameter is met.
….	…..	…...

Table 6: Equivalence between two syntactic arguments according to features_mapper and mapper.properties directives

2.1.3.3 Directives directives.txt and ignored.txt

These two directives tell the software the full mapping among values of the features attributes and which features must be ignored when two objects are compared respectively. The latter directive, usually, is used by the software to skip features: for example if the subject is always absent from one lexicon it should be skipped when two arguments are compared. The former contains the mapping among the values of features from input lexicons.

The format is value_lexicon_A (tab) value_lexicon_B and it is coherent with the two input lexicons, A and B:

#lexA (tab) lexB

Lexicon A	Lexicon B
aclauscomp adverbial aprepcomp clauscomp	complement
indirectobject	indirect_object
ncomp	complement

Table 7: mapping among values from different lexicons

We can see that the mapping is not limited to a 1 to 1 mapping, but also a N to M mapping is possible. The software checks all possible mapping before comparing objects.

2.1.3.4 Software flow

In this section we provide a brief outline of the mapping algorithm, for more information please see (R. Del Gratta et al., 2012). The algorithm consists of the following steps:

```
(Mapping) Reading and extracting the common verbs: Lexicons A and B are
 parsed; then a list of common verbs is created;
(Mapping) Extracting SCFs and SBs: For each common verb the tool extracts
 syntactic behaviours and connected SubCategorization Frames;
(Mapping) Comparing syntactic arguments: For each pair of SCFs, all
 Syntactic Arguments extracted and compared according to the directives in
 section 2.3.
(Mapping) Creating a report: the report ranks SCFs according to their
 similarity. In principle an user can edit this report and change the
 similarity among SCFs;
(Building) Reading and extracting the correct SBs: Lexicons A and B are
 parsed, the report is read and only those SCFs fully equivalent
 (similarity is 1) are extracted. From this list, the pointing SBs (from
 both lexicons) are compared using the directives in section 2.3
```

2.1.3.5 PANACEA Experiment

The experiment of merging two Italian SCFs lexicons has been carried out at ILC using the described software. The first lexicon is a subset of the PAROLE lexicon, (Ruimy et al., 1998), (a manually built lexicon); the second is a lexicon of SCFs automatically induced from a corpus.

The experiments has been replicated with two different set of parameters, see table 8. But only the second run of the experiment has been completed using also the building components.

run #1	run #2
th_min = 3 th_max = 3 sim min = 0.33 sim max = 1	th_min = 3 th_max = 3 sim_min = 1 sim_max = 1

Table 8: Directive parameters

The validation of the merger for this experiment is reported in D7.4 (Section 4.2.5) and in Del Gratta et al. (2012).

2.2 Multi-level Merging

In this section the multilevel merger created in the PANACEA project is described. With respect to the first one described in section 2.1, this version of the merger leaks the basic semantic interoperability, at least for the version released, but it embraces a full merging of main objects from the core and the morpho-syntax extension of LMF.

The code models a slightly revised DTD version 16 of the LMF2 model and adds some constraints: the main one is that all the ID attributes which in the DTD are defined IMPLIED are now defined REQUIRED. The original DTD is unchanged, but the tool creates the IDs when needed.

The reason behind this choice is that, in this way, the LMF structure resulting from the parsing of the LMF file, can be easily serialized into a database, IDs playing the role of primary and foreign keys.

The software is developed in Java and it is available as part of this deliverable.

2.2.1 General aspects

The merger manages two input lexical resources and returns a merged lexical resource. The ones in input, in principle, contain N and M lexicons respectively. The merger addresses this situation extracting all common lexical entries from the N+M input lexicons and generates the 1 lexicon with all common objects. The merger manages the N and M lexicons purging them of common objects, that's to say defining their complements.

2.2.2 Implemented LMF Structures

This chapter describes how DTD has been transformed into software objects and how these objects slightly modify the DTD itself.

[2] See http://www.tagmatica.fr/lmf/iso_tc37_sc4_n453_rev16_FDIS_24613_LMF.pdf

2.2.2.1 Lexical Resource From DTD

Orginal DTD (core)	Revised DTD (core)
<!ELEMENT LexicalResource (feat*, GlobalInformation, Lexicon+, SenseAxis*, TransferAxis*, ContextAxis*)> <!ATTLIST LexicalResource dtdVersion CDATA #FIXED "16">	<!ELEMENT LexicalResource (feat*, GlobalInformation, Lexicon+)> <!ATTLIST LexicalResource dtdVersion CDATA #FIXED "16">

Table 9: The DTD for Lexical resource

The current version of the software does not manage all original objects connected to the Lexical Resource class as defined in the LMF model.

2.2.2.2 Lexical Resource as Java Object

Full constructor with mandatory attributes subelements	Interfaces implemented

public LexicalResource(String dtdVersion,
GlobalInformation globalInformation, List<Lexicon> lexicons,
String **name**) {
……
addFeature(LMFAtts.__NAME__, name);
}
	IHasName
	Classes that implement this interface MUST have a feature whose attribute is name.

Table 10: The Java constructor for Lexical resource

The Lexical Resource is forced to have a human name because users can have an idea of what the lexical resource is about from its name.

The name attribute is used also for defining the lexical resource as a valid one. By default a lexical resource is valid if it contains one global information, one or more lexicons and the name. The attribute isvalid is crucial for the software since the procedures stops whether an input lexical resource is not valid.

2.2.2.3 Global Information From DTD and Java

The global information from DTD is a collections of features which contain various information. The Java object plainly reflects the DTD.

2.2.2.4 Lexicon From DTD

Orginal DTD (core)	Revised DTD (core)
<!ELEMENT Lexicon (feat*, LexicalEntry+, SubcategorizationFrame*, SubcategorizationFrameSet*, SemanticPredicate*,Synset*,SynSemCorrespondence*, MorphologicalPattern*, MWEPattern*, ConstraintSet*)>	<!ELEMENT Lexicon (feat*, LexicalEntry+, SubcategorizationFrame*,)>

Table 11: The DTD for Lexicon

The current version of the software does not manage all original objects connected to the Lexicon

2.2.2.5 Lexicon as Java Object

Full constructor with mandatory attributes and subelements	Interfaces implemented

public Lexicon(String id, String lang) {
....
addFeature(LMFAtts.__LANGUAGE__, **lang**);
}
	IHasIdentifier, IHasLanguageIdentifier
	Classes that implement these interfaces MUST have a feature whose attribute is **language** and which contains the iso code of the language the lexicon is about and an attribute which is the ID. If the ID is not in the original file (as usually is) then is created. The ID is necessary since one Lexical Resource can contain more than one lexicon.
	These interfaces are useful for serialization of the Lexicon into a database, the name of the Lexical resource and the IDs of the lexicons play the role of the natural key.

Table 12: The Java constructor for Lexicon

By default a lexicon is valid if it contains a not empty identifier and language. So far there is no control over the ISO-CODES of the languages3. The attribute *isvalid* is crucial for the software since the procedures stops whether a parsed lexicon is not valid.

2.2.2.6 Lexical Entry From DTD

The Lexical entry, from DTD, embraces various objects; since we have started from the Morphosyntax, only the corresponding objects have been implemented.

Orginal DTD (core)	Revised DTD (core)
<!ELEMENT LexicalEntry (feat*, Lemma, WordForm*, Stem*, ListOfComponents?,RelatedForm*, TransformCategory*, Sense*, SyntacticBehaviour*)> <!ATTLIST LexicalEntry id ID #IMPLIED morphologicalPatterns IDREFS #IMPLIED mwePattern IDREF #IMPLIED>	<!ELEMENT LexicalEntry (feat*, Lemma, WordForm*, ListOfComponents?, RelatedForm*, Sense*, SyntacticBehaviour*)> <!ATTLIST LexicalEntry id ID #IMPLIED

Table 13: The DTD for Lexical Entry

2.2.2.7 Lexical Entry as Java Object

The Lexical Entry as Java object needs the part of speech as mandatory feature. This is in line with the current encoding of LMF lexicons

[3] The software interface IHasLanguageIdentifier contains method signatures responsible for checking whether a supplied language cose is a valid ISO_369_3 code. Methods must be implemented in implementing classes. The current implementation returns always true.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public LexicalEntry(String pos, String id) {
this.pos = **pos**;
Id = id;
addFeature(LMFAtts.__POS__, **pos**);
}
	IHasIdentifier, IHasPos
	Classes that implement these interfaces MUST have a feature whose attribute is **partofspeech** and which contains the actual part of speech of the lexical entry and an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary since one Lexicon contain more than one lexical entry.
	These interfaces are useful for serialization of the Lexical entry into a database, the ID of the Lexicon and the IDs of the lexical entries play the role of the natural key.

Table 14: The Java constructor for Lexical Entry

By default a lexical entry is valid if it contains a not empty identifier and part of speech. The attribute *isvalid* is crucial for the software since the procedures skips non valid lexical entries.

2.2.2.8 Lemma From DTD

The Lemma is needed by the LMF model. In the tool it is used to connect the *writtenform* to the lexical entry. The lemma is also the first object of the morphological extension.

Orginal DTD (morpho)	Revised DTD (morpho)
<!ELEMENT Lemma (feat*, FormRepresentation*)>	<!ELEMENT Lemma (feat*)>

Table 15: The DTD for Lemma

The *formrepresentation* is an LMF object used to host the ways of writing entries. We had skipped this object and encoded the *writtenform* as a specific feature.

2.2.2.9 Lemma as Java Object

The Lemma as Java object needs the *writtenform* as mandatory feature.4. This is in line with the current encoding of LMF lexicons.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public Lemma(String id, String writtenForm) {
Id = id;
this.writtenForm = writtenForm;
}
	IHasIdentifier, IHasWrittenForm, IHasGrammaticalFeatures
	Classes that implement these interfaces MUST have a feature whose attribute is **writtenform** and which contains the canonical form of the lexical entry and an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary for serialization into database. There is a 1:1 relationship between lexical entries and lemmas: ID is also needed to establish this relation in a formal way.
	The Lemma, from the model, extends the Form which is a empty abstract object. This object implements interfaces which encode specific morphological features such the grammatical features. Lemma inherits this implementation and needs the grammatical features as well. However they are not mandatory

[4] In reality, the software as it is looks for writtenform in both lemma and lexical entry.

Full constructor with mandatory attributes and subelements	Interfaces implemented
	These interfaces are useful for serialization of the Lexicon into a database, the ID of the Lemma and the ID of the lexical entry play the role of the natural key.

Table 16: The Java constructor for Lemma

By default a lemma is valid if it contains a not empty identifier and a valid (not empty) *writtenform*. The attribute *isvalid* is crucial for the software since the procedures skips non valid lemmas

2.2.2.10 Wordform From DTD

Orginal DTD (morpho)	Revised DTD (morpho)
<!ELEMENT WordForm (feat*, FormRepresentation*)>	<!ELEMENT WordForm (feat*)>

Table 17: The DTD for Wordform

The *formrepresentation* is an LMF object used to host the ways of writing entries. We had skipped this object and encoded the *writtenform* as a specific feature of *WordForms*.

2.2.2.11 Wordform as Java Object

The *WordForm* as Java object needs *writtenform* as mandatory feature, check the note for lemma. This is in line with the current encoding of LMF lexicons.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public WordForm(String id, String writtenForm) {
Id = id;
this.writtenForm = writtenForm;
addFeature(LMFAtts.__WRITTENFORM__,
writtenForm);
}
	IHasIdentifier, IHasWrittenForm, IHasGrammaticalFeatures
	Classes that implement these interfaces MUST have a feature whose attribute is **writtenform** and which contains the canonical form of the lexical entry and an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary for serialization into database.
	The WordForm, from the model, extends the Form which is a empty abstract object. This object implements interfaces which encode specific morphological features such the grammatical features. Wordform inherits this implementation and defines the following: grammaticalNumber, grammaticalGender, grammaticalTense, person. However they are not mandatory
	These interfaces are useful for serialization of the Lexicon into a database, the ID of the Lexical entry and the IDs of the word forms play the role of the natural key.

Table 18: The Java constructor for Wordform

By default a *Wordform* is valid if it contains a not empty identifier and a valid (not empty) *writtenform*. The attribute *isvalid* is crucial for the software since the procedures skips non valid *WordForms*.

2.2.2.12 relatedform From DTD

Orginal DTD (morpho)	Revised DTD (morpho)
<!ELEMENT RelatedForm (feat*, FormRepresentation*)> <!ATTLIST RelatedForm targets IDREFS #IMPLIED>	<!ELEMENT RelatedForm (feat*, FormRepresentation*)> <!ATTLIST RelatedForm targets IDREFS #IMPLIED>

Table 19: The DTD for RelatedForm

We did not add any changes to the DTD.

2.2.2.13 RelatedForm as Java Object

The *RelatedForm* as Java object needs written form as mandatory feature, check the note for lemma. This is in line with the current encoding of LMF lexicons.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public RelatedForm(String id, String targets) {
Id = id;
this.targets = targets;
}
	IHasIdentifier, IHasWrittenForm, IHasGrammaticalFeatures
	Classes that implement these interfaces MUST have a feature whose attribute is **writtenform** and which contains the canonical form of the lexical entry and an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary for serialization into database.
	The Relatedform, from the model, extends the Form which is a empty abstract object. This object implements interfaces which encode specific morphological features such the grammatical features. RelatedForm inherits this implementation and needs the grammatical feaatures as well. However they are not mandatory.
	These interfaces are useful for serialization of the Lexicon into a database, the ID of the Lexical entry and the IDs of the related forms play the role of the natural key.

Table 20: The Java constructor for Relatedform

By default, a *RelatedForm* is valid if it contains a not empty identifier and a valid (not empty) *writtenform*. The attribute *isvalid* is crucial for the software since the procedures skips non valid *relatedforms*.

The software looks for possible "orphan" *relatedforms*, that's to say related forms whose targets attribute points to a not existent lexical entries.

2.2.2.14 ListofComponent and Components From DTD

Orginal DTD (multiword)	Revised DTD (multiword)
<!ELEMENT ListOfComponents (feat*, Component+)> <!ELEMENT Component (feat*)> <!ATTLIST Component entry IDREF #REQUIRED>	<!ELEMENT ListOfComponents (feat*, Component+)> <!ELEMENT Component (feat*)> <!ATTLIST Component entry IDREF #REQUIRED>

Table 21: The DTD for Component

We did not add any changes to the DTD.

2.2.2.15 Components5 as Java Object

The Component as Java object needs a valid pointed entry: a lexical entry which is in the lexicon.

Full constructor with mandatory attributes and subelements	Interfaces implemented
public Component() { }	No interfaces.

Table 22: The Java constructor for Component

Component only have the basic constructor, since the referenced target lexical entry has to be fixed once the lexical resource has been fully parsed. We have planned the *partofspeech* as a possible feature but not mandatory. Reasons behind are related to the multiword patterns that we have not been coded yet.

The software looks for "orphan" *components*, that's to say *components* that point to a nonexistent lexical entry in the lexicon.

2.2.2.16 SyntacticBehaviour From DTD

Orginal DTD (syntax)	Revised DTD (syntax)
<!ELEMENT SyntacticBehaviour (feat*)> <!ATTLIST SyntacticBehaviour id ID #IMPLIED senses IDREFS #IMPLIED subcategorizationFrames IDREFS #IMPLIED subcategorizationFrameSets IDREFS #IMPLIED>	<!ELEMENT SyntacticBehaviour (feat*)> <!ATTLIST SyntacticBehaviour id ID #IMPLIED subcategorizationFrames IDREFS #IMPLIED >

Table 23: The DTD for SyntacticBehaviour

We did not implement the relation between syntactic behaviours and senses, which is implemented at lexical entry level, instead.

2.2.2.17 SyntacticBehaviour as Java Object

The Syntactic Behaviour as Java object needs a valid pointed entry: a subcategorization frame which is in the lexicon.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public SyntacticBehaviour(String id, String idSCF, String aux, String corpusLabel) {
Id = id;
IdSCF = idSCF;
…....
}
	IHasIdentifier
	Classes that implement this interface MUST have an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary since one LexicalEntry contains more than one syntactic behaviour.
	The interface is useful for serialization of the syntactic behaviour into a database, the ID of the LexicalEntry and the IDs of the syntactic behaviours play the role of the natural key.

Table 24: The Java constructor for SyntacticBehaviour

[5] The *listofcomponents* object is not so interesting since it is just a list of components plus additional features.

The constructor of the *SyntacticBehaviour* contains two additional features, namely the auxiliary and the *corpusLabel*. The auxiliary is a property of the realization of the verb which differentiates verbs with the same subcategorization frame, for example in Italian "sono (ho) dovuto restare". The *corpusLabel* is used for the specific PANACEA merging scenario. The software looks for "orphan" syntactic behaviors, that's to say syntactic behaviors that point to a non-existent subcategorization frame in the lexicon.

2.2.2.18 SubcategorizationFrame From DTD

Orginal DTD (syntax)	Revised DTD (syntax)
<!ELEMENT SubcategorizationFrame (feat*, LexemeProperty?, SyntacticArgument*)> <!ATTLIST SubcategorizationFrame id ID #IMPLIED inherit IDREFS #IMPLIED>	<!ELEMENT SubcategorizationFrame (feat*, LexemeProperty?, SyntacticArgument*)> <!ATTLIST SubcategorizationFrame id ID #IMPLIED>

Table 25: The DTD for SubcategorizationFrame

We did not implement the frameset object.

2.2.2.19 SubcategorizationFrame as Java Object

The *SubcategorizationFrame*, as Java object needs a valid pointing entry: a syntactic behavior which points to this SCF.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public SubCategorizationFrame(String id) {
this.id=id;
}
	IHasIdentifier
	Classes that implement this interface MUST have an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The ID is necessary since one Lexicon contains more than one subcategorization frame
	The interface is useful for serialization of the syntactic behaviour into a database, the ID of the LexicalEntry and the IDs of the syntactic behaviours play the role of the natural key.

Table 26: The Java constructor for SubcategorizationFrame

The constructor of the Subcategorization frame needs the identifier of the subcategorization frame. This is used by the software for looking for "unused" subcategorization frames, that's to say SCFs which are never used by syntactic behaviours.

2.2.2.20 LexemeProperty From DTD

Orginal DTD (syntax)	Revised DTD (syntax)
<!ELEMENT LexemeProperty (feat*)>	<!ELEMENT LexemeProperty (feat*)>

Table 27: The DTD for LexemeProperty

We did not add changes to the *lexemeproperty* object.

2.2.2.21 LexemeProperty as Java Object

The *Lexemeproperty* as Java object is a simple collection of features.

Full constructor with mandatory attributes and subelements	Interfaces implemented
public Lexemeproperty() { }	**IHasIdentifier** Classes that implement this interface MUST have an attribute which is the ID. If the ID is not in the original file (as it could be) then is created. The interface is useful for serialization of the lexeme property into a database, the ID of the Subcatframe and the ID of the lexeme property play the role of the natural key.

Table 28: The Java constructor for LexemeProperty

The constructor of the *Lexemeproperty* is void.

2.2.2.22 SyntacticArgument From DTD

Orginal DTD (syntax)	Revised DTD (syntax)
<!ELEMENT SyntacticArgument (feat*)> <!ATTLIST SyntacticArgument id ID #IMPLIED target IDREF #IMPLIED>	<!ELEMENT SyntacticArgument (feat*)> <!ATTLIST SyntacticArgument id ID #IMPLIED>

Table 29: The DTD for SyntacticArgument

We did not implement the target, that's mean the Syntactic Arguments never point to Subcategorization frames (as it could be possible according to the rev. 16 model).

2.2.2.23 SyntacticArgument as Java Object

The *SyntacticArgument* as Java object is a simple collection of features which implement the syntactic properties of the specific argument.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public SyntacticArgument(String id) {
this.id=id;
}
	IHasIdentifier
	Classes that implement this interface MUST have an attribute which is the ID. If the ID is not in the original file (as it could be) then it is created.
	The interface is useful for serialization of the syntactic arguments into a database, the ID of the Subcatframe and the IDs of the syntactic arguments play the role of the natural key.

Table 30: The Java constructor for SyntacticArgument

The constructor of the Syntactic Argument takes only the id.

2.2.2.24 Sense From DTD

Orginal DTD (semantic)	Revised DTD (semantic)
<!ELEMENT Sense (feat*, Sense*, Equivalent*, Context*, SubjectField*, PredicativeRepresentation*, SenseExample*, Definition*, SenseRelation*, MonolingualExternalRef*)> <!ATTLIST Sense id ID #IMPLIED synset IDREF #IMPLIED>	<!ELEMENT Sense (feat*)> <!ATTLIST Sense id ID #IMPLIED >

Table 31: The DTD for Sense

We did not implement the synset neither the self relation among senses.

2.2.2.25 Sense as Java Object

The Sense as Java object is under development. So far a simple collection of features is implemented. Future versions of the merger will address semantic-related merging procedures, so that the sense will be expanded to cover sub elements as described in the DTD rev. 16.

Full constructor with mandatory attributes and subelements	Interfaces implemented

public Sense(String id) {
this.id=id;
}
	IHasIdentifier
	Classes that implement this interface MUST have an attribute which is the ID. If the ID is not in the original file (as it could be) then it is created.
	The interface is useful for serialization of the Sense into a database, the ID of the Lexical Entry and the IDs of the senses play the role of the natural key.

Table 32: The Java constructor for Sense

The constructor of the Sense takes only the id.

2.2.3 Multilevel Merging two Morphosyntactic lexicons

This chapter describes how the software deals with the merging of two lexicons. The released version of the software merges the following objects:

- Features
- Lexical Resource
- Global Information
- Lexicons
- Lexical entries
- WordForms
- RelatedForms
- Components
- Syntactic Behaviours

- Subcategorization Frames
- Syntactic Arguments
- Lexemeproperties
- Senses

The same version suffers of a limitation: namely the attributes and values in the input lexicons must be the same. On the contrary of the merger described in chapter 2, the current version of the tool does not implement the basic semantic interoperability. Next version of the tool will address this issue.

2.2.3.1 When are two object equivalent?

Figure 4 below shows how two objects, namely $object_a$ and $object_b$ can be considered as equivalent.

One lexical object, when formalized in a DTD, can be described as a list of attributes and properties, i.e. features. The equivalence between these two objects can be established at different levels and according to various needs. Some scenarios can require a full equivalence (all attributes and features are equivalent in both lexicons. In reality features are the same in both the lexicons: they have the same attribute and the same value.), while other can require a weaker equivalence, that's to say an equivalence where only some features are the same as well as only some attributes.

The challenge, here, is how to formalize all possible equivalences. It is clear that the tool has to follow human "directives" in and before taking decisions.

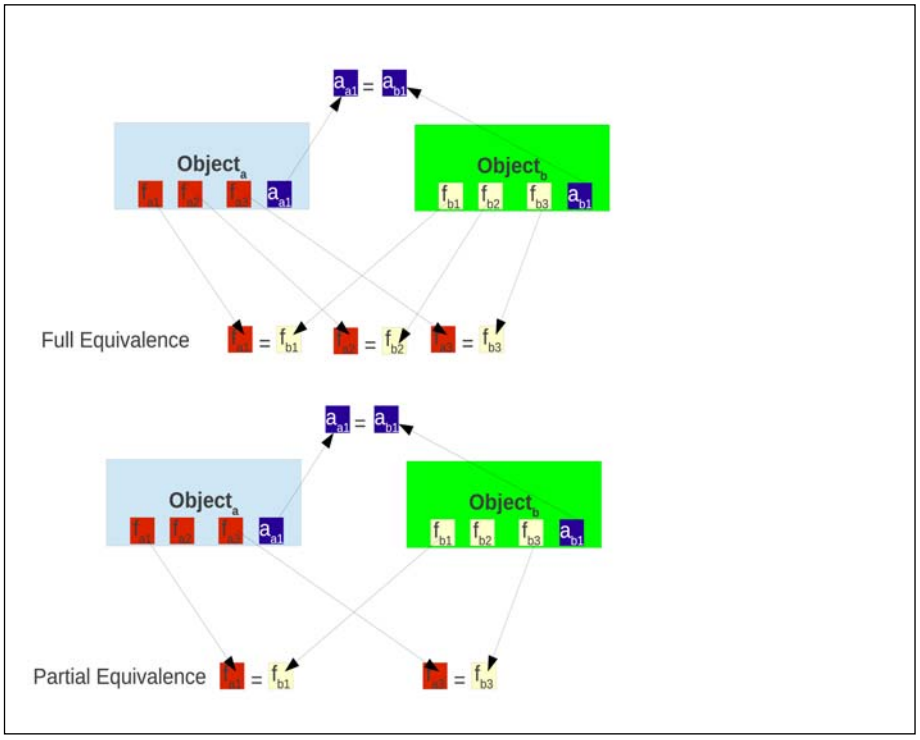

Figure 3: Full and partial equivalence

2.2.3.2 The equivalence rule directive

The rules to define equivalence among lexical objects has been coded by two distinct directive files: a file which contains "what to check" and a file which specifies the "features to check".

The first file describes what the tool has to check when it defines two objects as equivalent. Following the official documentation of the LMF rev. 16, lexical objects are adorned with certain features which add information to the objects which is summed up to the information possibly contained in the attributes. This file contains an entry *checkFeats* which tells the tool which features must be taken into account for the equivalence. These features are listed in the second file.

Figure 5 below describes the relations among DTD, Java and equivalence directive. The method *isEquivalent*(...) is defined for every object described in section 2.2. This method takes an integer as input parameter and switches to the correct equivalence method defined by the integer.

The integer is a value defined by the user which assign a bit to the things (attributes and features) to check. The tool reads the file and assigns the value to the corresponding lexical object.

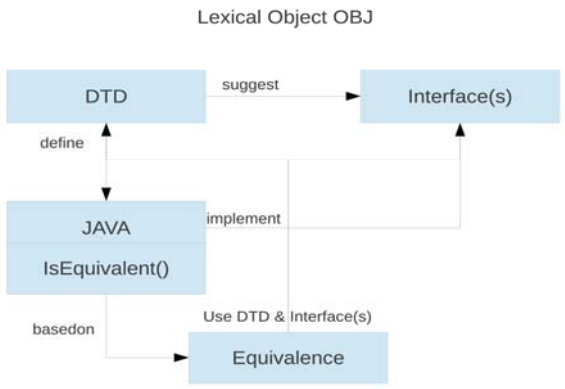

Figure 4: Relations among DTD, Java and equivalence rules

2.2.3.3 The equivalence rule by example

Following examples can help to understand the equivalence rule formalism:

Features can be defined equivalent if they have same attribute, same value or both. Attribute and value are listed in order of decreasing importance: *attribute,value*. Reading from lower to higher bit (right to left), the value has position 0, the attribute position 1. The three possible combination can be written as follows: "*,value*" ,"*attribute,*" and "*attribute,value*"; but a bitmap of the three possibilities is also possible:

$2^1=2$	$2^0=1$
Attribute (position 1)	Value (position 0)
[0|1] *0 is absent, 1 present	[0|1] *0 is absent, 1 present

Table 33: attributes and values assume numeric values according to the position

The three combination can be bitmapped as reported in table 35 and then transformed back to decimal values to be passed to the *isEquivalent()* method.

Combination	Bit sequence	Decimal value
,value	01	0*21+1*20=1
attribute,	10	1*21+0*20=2
attribute,value	11	1*21+1*20=3

Table 34: Combination → bitmap → decimal values

The "what to check" file contains, for features, the following instruction (lines starting with # are comments)

```
#          value      - position 0          2^0=1
#          attribute  - position 1          2^1=2
#          range 1 - 3
Feature = 3
```

Table 35: Equivalence rule for features

Setting "Feature =3" basically tells the method that the feature objects shall be considered equivalent iff both the attribute and value are the same. For other lexical objects the control *checkFeats* has been added in the lowest position, lowest bit.

```
#LexicalEntry
#checkFeats - position 0       2^0=1
#LexicalEntry = 0 for not CHECKING FEATS
LexicalEntry = 1

#WordForm
#checkFeats - position 0       2^0=1
#WordForm = 0 for not CHECKING FEATS
WordForm = 1
```

Table 36: Rules for LexicalEntry and WordForm

Both LexicalEntry and WordForm lexical objects have no attributes to be checked, so that the integer sent to the *isEquivalent()* method is either 0 or 1. If the value is 0, the tool does not check any features. On the other hand if the value is 1, the the list of features is extracted from the "features to check" file, as reported in table 38 below:

```
#The format of the file is
#Object = value
#LexicalEntry
LexicalEntry:partOfSpeech
LexicalEntry:writtenForm

#WordForm
#		personNumber
#		tense
#		verbMood
#		grammaticalGender
#		grammaticalNumber
#		writtenform
WordForm:personNumber
WordForm:tense
WordForm:verbMood
WorForm:grammaticalGender
WordForm:grammaticalNumber
WordForm:writtenform
```

Table 37: Features for lexical entries and wordforms

The tool assigns to the lexical objects the list of features:

LexicalEntry → *partOfSpeech,writtenForm*

WordForm → *personNumber,tense,verbMood,grammaticalGender,grammaticalNumber,writtenform*

The presented example involve features of the objects. But the same logic applies for attributes only and for combination of attributes and features. A typical example is the *SyntacticBehaviour*. Generally speaking two SBs can be considered equivalent if the point to equivalent SCFs and have the same auxiliary. The first is an attribute (in the DTD), while the second is a feature.

```
#SyntacticBehaviour
#		checkFeats		- position 0		2^0=1
#		idSCF			- position 1		2^1=2
#		range 1 - 3
#SyntacticBehaviour = 2 for not CHECKING FEATS.
#Check the attribute idSCF and feats
SyntacticBehaviour = 3
```

Table 38: Rule for SBs

The list of features to check is the following:

```
#SyntacticBehaviour
#           auxiliary  - position 0          2^0=1
#           id                    - position 1
            2^1=2
#           corpusLabel           - position 2
            2^2=4
#           idSCF                 - position 2
            2^3=8


SyntacticBehaviour:aux
SyntacticBehaviour:corpusLabel
```

Table 39: Features to control for SBs

The equivalence rule for SCFs is quite complex. This rule is based on the one defined for Syntactic arguments: two SCFs are equivalent if they have the same argument structure. Arguments are checked according to a list of features. Users can also add the *lexemeproperty* to the rule. In this case the tool selects among SCFs, that are equivalent for SAs, the ones that share the same *lexemeproperty*.

```
#SubCategorizationFrame
#           checkFeats          - position 0          2^0=1
#           id                  - position 1          2^1=2
#           Synargs             - position 2          2^2=4
#           lexemeproperty      - position 3          2^3=8
#           range 1 - 15
#SubCategorizationFrame = 4 for not CHECKING FEATS. SCF has no essential
features to check
SubCategorizationFrame = 4


#SyntacticArgument
#           checkFeats          - position 0          2^0=1
#SyntacticArgument = 0 for not CHECKING FEATS
SyntacticArgument = 1


#LexemeProperty
#           checkFeats          - position 0          2^0=1
#LexemeProperty = 0 for not CHECKING FEATS
LexemeProperty=1
```

Table 40: Equivalence rule for SCFs

The equivalence rule for LexemeProperty is standard, a list of features, but the equivalence rule for SyntacticArguments needs additional explanations. For some reasons connected to the automatic extraction methods of SCFs, it may happen that some features and/or values of features are not extracted[6]. To address this situation, new instructions have been added to the "features to check" file.

```
# Part of  things to exclude in checking
!SyntacticArgument:realization=*
!SyntacticArgument:introducer=da
!SyntacticArgument:=env

#!SyntacticArgument:function=*
# Objects to exclude in checking
#-SyntacticArgument:realization=*
-SyntacticArgument:function=subject
#-SyntacticArgument:function=object
```

Table 41: Features and/or values to skip

Where the syntax of the instructions is as follows. The "**!**" tells the tool that the following lexical object, in the example the *SyntacticArgument*, has some features which are to be skipped.

The "**:**" is the separator between the lexical object and the features to exclude.

This syntax accomplishes for punctual exclusion such in *introducer=da;* in this case the tool will exclude the corresponding feature applying **Feature=3** equivalence mode; for excluding all features whose value is *env* (applying the **Feature=1** equivalence mode); for excluding all features whose attribute is function such as *function=** (applying **Feature=2** equivalence rule). In all these examples the tools manages features accordingly. For example *realization=np* and *realization=pp* are considered equivalent.

The second part of table 42 addresses the full exclusion of objects. The "**-**" tells the tool that the following lexical object, in the example the *SyntacticArgument*, has to be removed when 2 Syntactic arguments are checked. The "**:**" is the separator between the lexical object and the complete features to exclude.

The rule "*SyntacticArgument:function=subject*" tells the tool to exclude the SA which contains such feature.

The exclusion rules are applied by the tool when two SCFs are compared:

Exclusion rule	**SCF-SA$_a$**	**SCF-SA$_b$**	**Equivalent**
!SyntacticArgument:realization=*	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization**" val="**np**"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="subj"/> <feat att="**realization**" val="**pp**"/> <feat att="introducer" val="to"/> </Syn....>	YES(1)

[6] For example the Italian SCF lexicon, automatically extracted, does not list the function=subject feature. In this case SAs with this feature will never match with SAs that do not have this feature, even if all other features (introducer, realization) are equivalent. This situation should be addressed.

Exclusion rule	SCF-SA$_a$	SCF-SA$_b$	Equivalent
-SyntacticArgument:function=subject	<Syn.... > <feat att="**function" val="subject**"/> <feat att="realization" val="np"/> <feat att="introducer" val="to"/> </Syn....> <Syn.... > <feat att="function" val="indirect-object/> <feat att="realization" val="np"/> <feat att="introducer" val="to"/> </Syn....>	<Syn.... > <feat att="function" val="indirect-object"/> <feat att="realization" val="np"/> <feat att="introducer" val="to"/> </Syn....>	YES(2)

Table 42: Exclusion rules for SAs and SCFs

(1) The first two SAs are equivalent because the two different features *realization=np* and *realization=pp* are considered as the same. (2)The first structure contains two SAs, while the second has only one SA. These two structures are equivalent since the equivalence rule applied tells the software to cut the SA which contains the feature *function=subject.*

2.2.4 Merger WorkFlow

This section describes the general workflow of the merger. The main workflow is reported in figure 5. The tool takes two lexical resources in input; analyzes all possible lexicons for each lexical resource, does "something" with the lexical objects and produces a merged lexical resource in output.

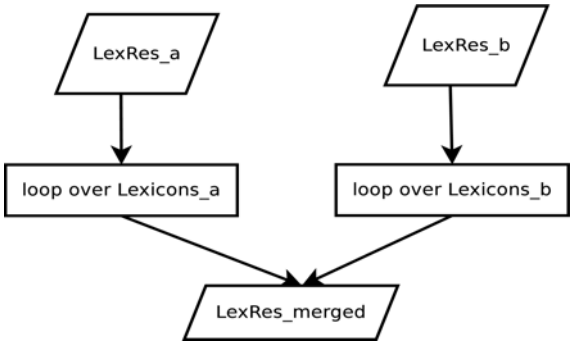

Figure 5: Basic Workflow

2.2.4.1 Detailed workflow

In this section the detailed workflow, reported in figure 6, is described. The merger parses the two lexical resources and creates a software data structure which reflects the lexical resources.

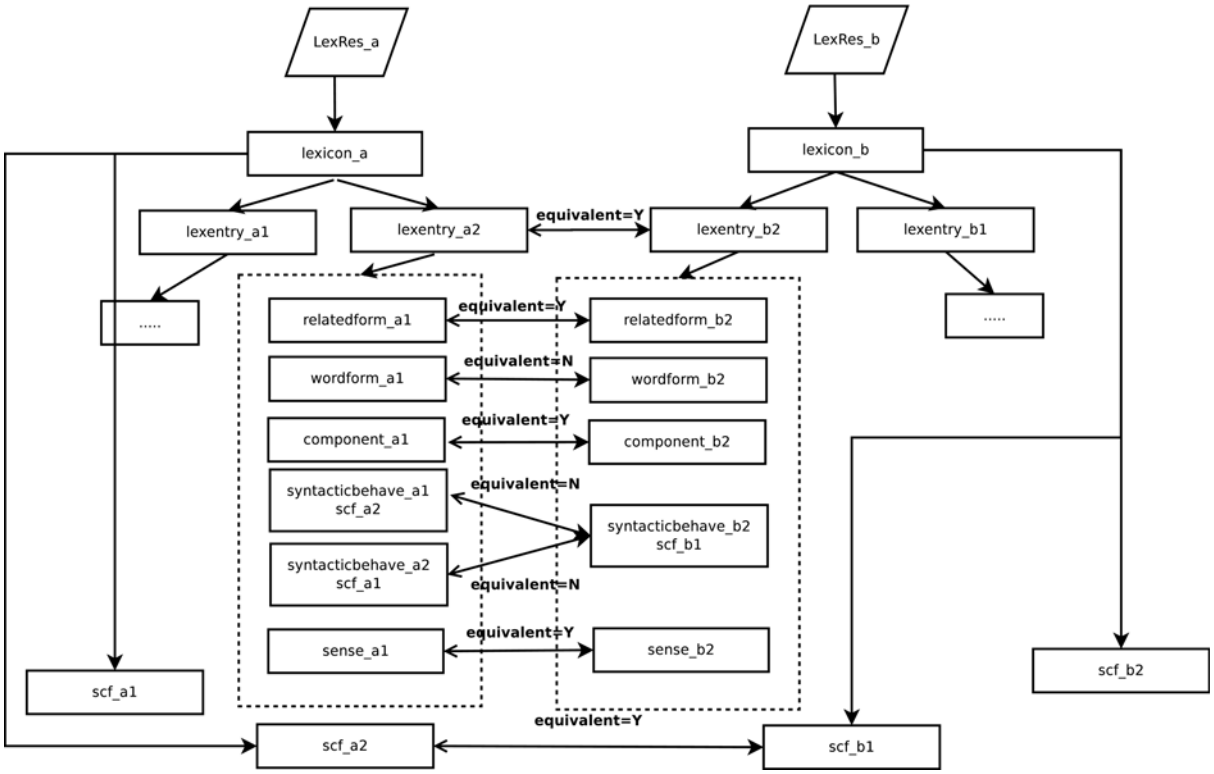

Figure 6: Detailed WorkFlow

There are two main building blocks that the software analyzes:

- The *SubcategorizationFrames* for the syntax
- The *LexicalEntry* for morphology and for link to the syntax (*SyntacticBehaviours*)

The tool executes the following steps:

```
1. For each lexicon contained in both lexical resources the lexical entries are
   extracted and those coming from lexicons in lexical resource (a) are checked
   against the ones from lexical resource (b);
2. For each lexical entry in (a) which is equivalent (according to the specific
   rule) to a lexical entry in (b) all sub elements are extracted;
3. For each subelement extracted in 2 in (a) and in (b) the equivalence is
   checked;
4. For each lexicon contained in both lexical resources the SCFs are extracted
   and those coming from lexicons in lexical resource (a) are checked against
   the ones from lexical resource (b);
5. For each SCF in (a) which is equivalent (according to the specific rule) to
   a SCF in (b) the pointing Syntacticbehaviours are extracted and ids are
   managed;
6. The output is written.
```

Point 5 is interesting. From figure 6 the SyntacticBehaviour *syntacticBehave_a1* points to *scf_a2* and *scf_a2* is equivalent to *scf_b1* which is, in turn, pointed by *syntacticBehave_b1*. But these two syntactic behaviours are not equivalent according to their equivalence rule. In this case the tool assigns to the *lexicalentry_a2* both syntactic behaviours. On the other side the merged lexical resource will contain only the *scf_a2* subcategorization frame:

Lexicon lexicalentry_a2 syntacticBehave_a1 scf=**scf_a2** syntacticBehave_b1 scf=***scf_b1*** **scf_a2**

Table 43: The merged lexical resource

The lexical resource is inconsistent, since the *syntacticbehave_b1* is orphan. The tool manages this issue by rewriting all ids of objects which are linked to common lexical objects[7]:

Lexicon lexicalentry_a2 syntacticBehave_a1 scf=scf_a2 syntacticBehave_b1 scf=scf_a2 scf_a2

Table 44: The merged lexical resource after id management

2.2.5 Differences between the two mergers

This chapter summarizes the two mergers.

SubCatFrame oriented characteristics	**Morpho-Syntax oriented characteristics**
Extract common LexicalEntries; Focused on SubCatFrames (SCFs) only. Manages similarity of SCFs according to a cosine similarity (CS) among Syntactic Arguments (Sas); Merge Syntactic Behabiours (SBs) of totally equivalent SCFs (CS=1); User driven via directive files;	Extract common LexicalEntries; Focused on the MorphoSyntax core+extension. Manages similarity of SCFs according to a cosine similarity (CS) among Syntactic Arguments (Sas); Merge features of every single lexical object: lexical resource, global information, lexentries....; Merge lexical objects related to common lexentries: wordforms, relatedforms, components (MW), SB, SCFs, Sas; Manages IDs and orphans/unused objects; User driven via equivalence rules.

[7] These objects are syntactic behaviour, related forms and components. All these objects have IDREF in the DTD, so they need to point to objects that are in the same lexicon.

SubCatFrame oriented characteristics	**Morpho-Syntax oriented characteristics**
Work on common Lexical entries ? YES Generic? NO (Only SCFs and related SBs are managed) Implement semantic interoperability? YES (A basic semantic interoperability is set up for SAs. Namely attributes and values are reciprocally mapped) User driven ? YES via directive files; Modular in output lexicon(s) NO Only the intersection of two lexicons is outputted.	Work on common Lexical entries ? YES Generic? NO (Only morpho syntactic extension and the Sense object) Implement semantic interoperability? NO User driven ? YES via equivalence rule; Modular in output lexicon(s) YES For a two lexical resources in input which contain N and M lexicons respectively, the output is a lexical resources with N+M+1 lexicons: intersections and complements. Users can decide to compact the N+M+1 lexicons into only one.

Table 45: Synopsis of the two mergers

2.2.6 To do List

From table 46 can be noticed that the major weakness of the second merger is related to the non implementation of the basic semantic interoperability which is managed by the first merger. On the contrary the second one offers many other capabilities including the one for modeling the output. However, ILC has the commitment to add the semantic interoperability to the second merger.

2.3 Merging Lexicons with Graph Unification

UPF experimented to develop a new approach to fully automatically merge lexicons using graph unification. The research in this line has been conducted merging already existing lexicons. The proposed method has been tested in different scenarios with different formats and information. The main goal of the performed research was to develop a method to merge lexicons fully automatically.

Basically, the merging of lexicons has two well defined steps (Crouch and King, 2005):

1. Mapping Step: because information about the same phenomenon can be expressed differently, the information in the existing resources has to be extracted and mapped into a common format.
2. Combination Step: once the information in both lexicons is encoded in the same way, this information from both lexicons is mechanically compared and combined to create the new resource.

Thus, our goal was to carry out the two steps of the merging process in a fully automatic way. This is to perform both mapping and combination steps without any human supervision.

Necsulescu et al. (2011) and Bel et al. (2011) proposed to perform the combination step using graph unification (Kay, 1979). This single operation which is based on set union of compatible feature values, makes it possible to validate the common information, exclude the inconsistent one and to add, if desired, the unique information that each lexicon contained for building a richer resource. For graph unification in our experiments, we used the NLTK unification mechanism (Bird, 2006). This mechanism proved to be useful for building richer lexicons, but the most difficult part was to convert both lexicons into a common format

that allows the merging, this is, the mapping step.

Thus, a method to avoid manual intervention when converting two lexicons into a common format with a blind, semantic preserving method was devised (Bel et al., 2011). This proposal departs from the idea of Chan and Wu (1999) of comparing information contained in common entries of different lexicons and looking for significant equivalences in terms of consistent repetition. The basic requirement for this automatic mapping is to have a number of common entries encoded in the two lexicons to be compared. Chan and Wu (1999) were working only with single part-of-speech tags, but the lexicons we address here handle more complex and structured information, which has to be identified as units by the algorithm. In order to avoid the necessity of defining the significant pieces of information to be mapped by hand, we proposed a method to first automatically identify such pieces ("minimal units") in each lexicon and secondly, to automatically learn the correspondence of such pieces between the two lexicons. The results showed that it is possible to assess that a piece of the code in lexicon A corresponds to a piece of code in lexicon B since a significant number of different lexical entries hold the same correspondence. Then, when a correspondence is found, the relevant piece in A is substituted by the piece in B, performing the conversion into the target format to allow for comparison and, merging. Note that the task is defined in terms of automatically learning correspondences among both, labels and structure since both may differ across lexicons.

This technique has been applied in two different scenarios: on the one hand two subcategorization frame (SCF) lexicons for Spanish have been merged into one richer lexical resource. On the other hand, two morphological dictionaries were merged. In both cases the original lexicons were manually developed.

Regarding the merging of SCF lexicons, the two original SCF lexicons were developed for rule-based grammars: the Spanish working lexicon of the Incyta Machine Translation system (Alonso, 2005) and the Spanish working lexicon of the Spanish Resource Grammar, SRG, (Marimon, 2010) developed for LKB framework (Copestake, 2002). The SRG lexicon was already encoded as feature structures, but Incyta lexicon was encoded as a parenthesized list of all the possible subcategorization patterns. The conversion into a common format proved to be complicated due to the non-standard encoding of the lexicons, but feasible, achieving up to 92% in precision and 93% in recall when comparing automatically to manually extracted entries. The details of this experiment can be found in Bel, Padrò, Nesculescu (2011), Necsulescu et al. (2011) and Padró, Bel, Nesculescu (2011).

In the second experiment we extended and applied the same technique to perform the merging of morphosyntactic lexicons encoded in LMF. Lexical Markup Framework, LMF (Francopoulo et al. 2008) is an attempt to standardize the format of computational lexicons and may be useful to reduce the complexities of merging lexicons. However, LMF (ISO-24613:2008) "does not specify the structures, data constraints, and vocabularies to be used in the design of specific electronic lexical resources". Therefore, the merging of two LMF lexicons is certainly easier, but only if both lexicons also share the structure and vocabularies, if not, mapping has still to be done by hand or automatically.

The aim of this second experiment was to assess to what extent the actual merging of information contained in different LMF lexicons could be done automatically, following the proposed method, in two cases: when the lexicons to be merged share structure and labels, and when they do not. Besides, our second goal was to prove the generality of the approach, i.e. if it could be applied to different types of lexical resources.

Therefore, we applied the method presentedbefore to merge different Spanish morphosyntactic dictionaries. A first experiment tackled the merging of a number of dictionaries of the same family: Apertium monolingual lexicons (Armentano-Oller et al. 2007) developed independently for different bilingual MT modules. A second experiment merged the results of the first experiments with the Spanish morphosyntactic

FreeLing lexicon (Padró et al. 2010). All the lexicons were already in the LMF format, although Apertium and FreeLing have different structure and tagset. In addition, note that these morphosyntactic lexicons contain very different information than SCF lexicons of the first experiment, so this second scenario can be considered a further proof of the good performance and generality of the proposed automatic merging method.

The results of that work showed that the availability of the lexicons to be merged in a common format such as LMF indeed alleviates the problem of merging. In our experiment with different Apertium lexicons it was possible to merge three different monolingual morphosyntactic lexicons with the method proposed as to achieve a larger resource. We have also obtained good results in automatically converting to a common format and merging Apertium and FreeLing lexicons, this is. two LMF lexicons with different tag set. Details about these experiments can be found in Padró, Bel, Nesculescu (2012).

3 Lexical Merger Web Services

The list of deployed web services for lexical merger is presented in this section. There are currently three workflows which accomplish the merging of two acquired lexicons. Additional web services will be released in T32, especially related to multi-level merging.

Name	Type	Category	Language	Provider	Registry Number
lmf_merger	Soaplab			CNR-ILC	251
merge_lmf_files	Soaplab			UPF	245
merge_list_of_lmf_files	Soaplab			UPF	270
lmf_ml_merger	Soaplab			CNR-ILC	300

3.1 lmf_merger

http://registry.elda.org/services/251

Given two LMF files, this web service merges them into a single LMF file.

It can manage LMF files encoding the information in different ways as well as in the same way: users can supply their directives to the service. This will work, for example, for merging different lexicons learnt under PANACEA platform, i.e. which will be encoded in the same way; but the service is able to merge "gold" lexicon with automatically extracted ones. For example, when the PAROLE Italian Lexicon which is created by linguists is merged with an extracted lexicon.

This version of the webservice only works for LMF files which contain "<*LexicalEntry*>", "<*SyntacticBehaviours*>" and "<*SubcategorizationFrame*>" elements. The input of the service consists of two lexicons and four directives used by the service to map features and values; the output of the service is a single LMF file containing the intersection, common "<*LexicalEntry*>", of the two original lexicons.

3.2 merge_lmf_files

http://registry.elda.org/services/245

Given two LMF files, this web service merges them into a single LMF file. It works for LMF files encoding the information in the same way, i.e. same labels, values and structure. This will work, for example, for merging different lexicons learnt under PANACEA platform. If the LMF files contain equivalent information encoded in different ways, a mapping into a common format should be previously performed.

This version of the webservice only works for LMF files without references. This is, LMF files containing only "<*LexicalEntry* >" elements. I.e. it works for morphological dictionaries, noun classification, etc but not for SCF lexicons, for example.

The input of the service are two LMF files, and the output a single LMF file containing information from the two original lexicons.

3.3 merge_list_of_lmf_files

http://registry.elda.org/services/270

Given a list of URLs pointing to LMF files, this webservice merges them into a single LMF file. This is an extension of the merge_lmf_files, so it works for the same kind of lexicons.

The input of the service is a list of URLs pointing to the LMF files to be merged. The output is a single LMF file with all information combined.

3.4 lmf_ml_merger

http://registry.elda.org/services/300

This service applies the multi level merger described above. The service takes two lexical resources and two directive files as well as other parameters, as explained in table 3.1.

The two input lexical resources can, in principle, contain N and M lexicons respectively. The merger addresses this situation extracting all common lexical entries from the N+M input lexicons and generates the 1 lexicon with all common objects. The merger manages the N and M lexicons purging them of common objects, that's to say defining their complements.

Parameter	Comment
first_lmf	This is the first LMF file. The file contains Morpho-syntax objects plus the sense.
second_lmf	This is the second LMF file. The file contains Morpho-syntax objects plus the sense.
what2check	This file contains which attribute of specific lexical objects should be taken into account when those objects are compared. It contains also a *checkFeature* instruction. If this instruction is present, then the tool reads the features to check in the feats2check file. The format is the

Parameter	Comment

	following:
	#LexicalEntry
	#checkFeats - position 0 2^0=1
	#LexicalEntry = 0 for not CHECKING FEATS
	LexicalEntry = 1

feats2check	This file contains the list of features to check for a specific lexical object. The format is the following: #LexicalEntry LexicalEntry:partOfSpeech LexicalEntry:writtenForm
compact	This parameter is a bit [0|1] to tell the tool to produce the output lexical resource with only 1 lexicon which is the union of the intersection of the input lexicons plus the two complements or N+M+1 lexicons. Use 1 to compact, 0 to write N+M+1 lexicons. Default is 0
write_stat	This parameter is a bit [0|1] to tell the tool to write the cosine similarity measure among SCFs in the input lexicons. Use 1 to write statistics in the out_stats_mode output, 0 for skipping. Default is 1
out_stats_mode	is the output print stream where statistics are written.
out_file_mode	is the output print stream where the merged resource is written.

Table 46: Service parameters

4 Workflows

Here the workflows related to merging are presented. Workflows are also documented at myexperiment.elda.org. Additional workflows related to multi-level merger are pending.

4.1 Classification of nouns found in crawled data into lexical classes

http://myexperiment.elda.org/workflows/63

This workflow annotates with FreeLing the input crawled data (in TO1 format) and sends it to three different noun classifiers: event, location and human nouns. Each classifier produces a LMF output. The three obtained LMF files are merged into a single LMF lexicon containing information for all classes using the merging web service. This workflow works for English and Spanish, since those are the languages for which there are noun classifiers available.

Provider: UPF

4.2 Classification of nouns in PoS tagged data for English and 7 available classes

http://myexperiment.elda.org/workflows/84

This workflow uses FreeLing annotated data to classify the given list of nouns with the different available noun classifiers for Enlgish (7 classes). The LMF ouputs of each classifier are merged into a single LMF lexicon containing information for all classes.

Provider: UPF

4.3 Classification of nouns in PoS tagged data for Spanish and 9 available classes

http://myexperiment.elda.org/workflows/85

This workflow uses FreeLing annotated data to classify the given list of nouns with the different available noun classifiers for Spanish (9 classes). The LMF ouputs of each classifier are merged into a single LMF lexicon containing information for all classes.

4.4 Merging of two SCF LMF Lexicon

http://myexperiment.elda.org/workflows/72

This is a simple workflow for merging two existing lexicons of subcategorisation frames represented and encoded in LMF. In the PANACEA scenario, one lexicon is automatically produced through the platform and saved on the user local machine and the second lexicon is an already existing lexicon owned by the user, or retrieved/downloadable from third parties. This workflow only provides in input two LMF SCF lexicons and uses default directives for mapping and merging.

Provider: ILC

4.5 Merging of two Morpho-syntactic LMF lexicons

http://myexperiment.elda.org/workflows/95

This is a simple workflow for merging two existing lexicons which contain morpho syntactic objects including sense. This workflow applies the basic lmf_ml_merger, that's to say it creates a compacted lexicon

as output and does not provide statistics.

Provider: ILC

5 LREC 2012 Workshop

At LREC 2012 the PANACEA project held a successful and well-attended half-day workshop on "The Automatic Merging of Lexical Resources". The list of presentations is reproduced here:

Núria Bel, *Introduction to the Workshop*

Invited talk: Iryna Gurevych, *How to UBY – a Large-Scale Unified Lexical-Semantic Resource*

Oral Session:

Laura Rimell, Thierry Poibeau and Anna Korhonen, *Merging Lexicons for Higher Precision Subcategorization Frame Acquisition*

Muntsa Padró, Núria Bel and Silvia Necşulescu, *Towards the Fully Automatic Merging of Lexical Resources: A Step Forward*

Benoît Sagot and Laurance Danlos, *Merging Syntactic lexicons: The Case for French Verbs*

Cristina Bosco, Simonetta Montemagni and Maria Simi, *Harmonization and Merging of Two Italian Dependency Treebanks*

Poster Session:

Riccardo Del Gratta, Francesca Frontini, Monica Monachini, Valeria Quochi, Francesco Rubino, Matteo Abrate and Angelica Lo Duca, *L-Leme: An Automatic Lexical Merger Based on LMF Standard Technologies*

Anelia Belogay, Diman Karagiozov, Cristina Vertan, Svetla Koeva, Adam Przepiórkowski, Maciej Ogrodniczuk, Dan Cristea, Eugen Ignat and Polivios Raxis, *Merging Heterogeneous Resources and Tools in a Digital Library*

Thierry Declerck, Stefania Racioppa and Karlheinz Möhrt, *Automatized Merging of Italian Lexical Resources*

Radu Simionescu and Dan Cristea, *Towards a Universal Automatic Corpus Format Interpreter Solution*

The full proceedings can be found at this URL.

6 Documentation and Scientific articles

A full list of papers related to WP6 is given in D6.2, with the papers themselves annexed to D6.2. The following papers are relevant to the merging experiments and components:

LMF-based merging of SCF Lexica:

Del Gratta R., F Frontini, M Monachini, V Quochi, F Rubino, M Abrate, A. Lo Duca. 2012. L-LEME: an Automatic Lexical Merger based on the LMF Standard. In P*roceedings of LREC 2012 Workshop on Language Resource Merging.* Istanbul, Turkey.

Use of graph unification to merge two SCF lexicons after manually converting them into a common

format:

Silvia Necşulescu, Núria Bel, Muntsa Padró, Montserrat Marimon and Eva Revilla. 2011. Towards the Automatic Merging of Language Resources. In P*roceedings of WoLeR 2011*. Ljubljana, Slovenia.

Automatic conversion of the two SCF lexicons into a common format that allows posterior merging:

Muntsa Padró, Núria Bel and Silvia Necşulescu. 2011. Towards the Automatic Merging of Lexical Resources: Automatic Mapping. In *Proceedings of RANLP 2011*. Hissar, Bulgaria

Núria Bel, Muntsa Padró and Silvia Necşulescu. 2011. A Method Towards the Fully Automatic Merging of Lexical Resources. In *Proceedings of Language Resources, Technology and Services in the Sharing Paradigm Workshop.* November 12, 2011 at IJCNLP 2011 (Chiang Mai, Thailand).

Use of the method to merge morphosyntactic lexicons encoded in LMF:

Muntsa Padró, Núria Bel and Silvia Necşulescu. 2012. Towards the Fully Automatic Merging of Lexical Resources: a Step Forward. In P*roceedings of LREC 2012 Workshop on Language Resource Merging.* Istanbul, Turkey.

Merging for improving precision:

Rimell, Laura; Poibeau, Thierry and Korhonen, Anna. (2012). Merging Lexicons for Higher Precision Subcategorization Frame Acquisition. In *Proceedings of the LREC Workshop on Language Resource Merging*, Istanbul, Turkey.

7 References

Juan Alberto Alonso, András Bocsák. 2005. Machine Translation for Catalan-Spanish. The Real Case for Productive MT; In Proceedings of the tenth Conference on European Association of Machine Translation (EAMT 2005), Budapest, Hungary.

Carme Armentano-Oller, Antonio M. Corbí-Bellot, Mikel L. Forcada, Mireia Ginestí-Rosell, Marco A. Montava, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Gema Ramírez-Sánchez, Felipe Sánchez-Martínez. 2007. Apertium, una plataforma de código abierto para el desarrollo de sistemas de traducción automática. *In Proceedings of FLOSS (Free/Libre/Open Source Systems) International* Conference, p. 5-20. Jerez de la Frontera, Spain.

Núria Bel, Muntsa Padró and Silvia Necşulescu. 2011. A Method Towards the Fully Automatic Merging of Lexical Resources. In Proceedings of Language Resources, Technology and Services in the Sharing Paradigm, workshop in IJNLP 2011.

Steven Bird. 2006. NLTK: the natural language toolkit. In Proceedings of the COLING/ACL on Interactive presentation sessions. Association for Computational Linguistics, Morristown, NJ, USA.

Daniel K. Chan and Dekai Wu. 1999. Automatically Merging Lexicons that have Incompatible Part-of-Speech Categories. Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-99). Maryland.

Ann Copestake. 2002. Implementing Typed Feature Structure Grammars. CSLI Publications, CSLI lecture notes, number 110, Chicago.

Dick Crouch and Tracy H. King. 2005. Unifying lexical resources. Proceedings of Interdisciplinary Workshop on the Identification and Representation of Verb Features and Verb Classes. Saarbruecken; Germany.

Gil Francopoulo, Núria Bel, Monte George, Nicoletta Calzolari, Mandy Pet, and Claudia Soria. 2008. Multilingual resources for NLP in the lexical markup framework (LMF). Journal of Language Resources and Evaluation, 43 (1).

Montserrat Marimon. 2010. The Spanish Resource Grammar. Proceedings of the Seventh Conference on International Language Resources and Evaluation (LREC'10). Paris, France: European Language Resources Association (ELRA).

Silvia Necşulescu, Núria Bel, Muntsa Padró, Montserrat Marimon and Eva Revilla. 2011. Towards the Automatic Merging of Language Resources. In Proceedings of WoLeR 2011. Ljubljana, Slovenia.

Lluís Padró, Miquel Collado and Samuel Reese and Marina Lloberes and Irene Castellón. FreeLing 2.1: Five Years of Open-Source Language Processing Tools. Proceedings of 7th Language Resources and Evaluation Conference (LREC 2010), ELRA. La Valletta, Malta. May, 2010.

Muntsa Padró, Núria Bel and Silvia Necşulescu. 2011. Towards the Automatic Merging of Lexical Resources: Automatic Mapping. In Proceedings of RANLP 2011. Hissar, Bulgaria

Muntsa Padró, Núria Bel and Silvia Necşulescu. 2012. Towards the Fully Automatic Merging of Lexical Resources: a Step Forward. In Proceedings of LREC 2012 Workshop on Language Resource Merging. Istambul, Turkey.