Batch creation of Papillon entries from DiCo

Guy Lapalme
RALI-DIRO, Université de Montréal
Gilles Sérasset
GETA-IMAG, Grenoble

June 9, 2003

Abstract

This article describes our experience in creating in one batch about 700 linked Papillon entries by transforming DiCo entries. Although this conceptually simple task seems straightforward, it has raised a number of interesting questions about the structure of the Papillon dictionary. We describe the steps of this conversion process as well as one false start, hoping that this experience will be useful for other researchers of the Papillon project.

1 Introduction

When the first author came to Grenoble for a few months during summer and autumn 2002, his plan was to develop a text generator taking as input Papillon entries in order to help Papillon collaborators validate their input. As the complex relations and fields in a Papillon [7] entry can complicate the correct filling of entries, it was judged interesting to try to provide users some *linguistic* feedback by generating example sentences from linguistic informations already entered in the dictionary.

In November 2002, as there was only **one** valid (in the XML sense) Papillon entry, we decided to add more entries before building the example generator. In October 2002, François Lareau [3] had just finished his Master's Thesis in which, in the context of the `Sentence Garden` system, he provided `DiCo court` as basic data about 300 entries in the DiCo [6] formalism directly inspired by the DEC [5] of Igor Melčuk and his team. As the DEC has also strongly influenced the structure of Papillon entries, we felt that these DiCo entries could be used as a base for creating new Papillon entries.

We will now give an overview of both DiCo and Papillon entries and then we will describe how we managed to transform one into the other.

1.1 Sample DiCo entry

DiCo has been defined by Alain Polguère [6] based on a simple structure using a FileMaker Pro database. Figure 1 shows a sample entry as it is presented to a user who wants to consult, create or modify an entry. The only explicit structure is in the form of fields with possible internal line separators within some of them. The syntax of the fields is given by informal writing conventions defined by the team at the Université de Montréal. It is possible to get a text only version of each entry by exporting the database as a tab separated line (internal line separators are coded internally with another control character).

1.2 Sample XML Papillon entry

The Papillon design principles have been described in the Ph.D. thesis of Mathieu Mangeot-Lerebours [4]. Its format is defined as a hierarchy of XML Schemas whose fields, like DiCo's, also follow quite closely Meaning-Text Theory and the DEC. A Papillon entry must then be validated according to these schemas. The three levels of Schemas are the following:

- `dml.xsd` defines a general structure for any computer based dictionary
- `papillon.xsd` defines language-independent information for any Papillon dictionary
- `papillon_fra.xsd` defines language (in this case French) specific information

Figure 2 gives an example Papillon entry. It was produced by our system from the textual format shown in figure 1.

2 First try

As François Lareau had already developped, using Sicstus Prolog on Macintosh, a parser for validating DiCo entries and transforming them in Prolog clauses, we thought it would be simple to use the Prolog clauses as input for creating Papillon entries. Unfortunately, the fact that Sicstus Prolog was not really well integrated on the Mac and that the Unicode coding of characters needed for XML entries was not convenient in Prolog programs raised some technical difficulties. But more important was the fact that *Sentence Garden* dealt only with lexical functions and government patterns, so we would have had to parse the original DiCo file anyway. Given our previous experience in using a Java program to create an XML file, we thought it would be simple and straightforward to process the whole DiCo file in Java.

3 Creating XML Papillon entries in Java

The resulting Java program [2] (15 classes for about 1200 lines of code) proved to be more complex than expected because DiCo entries are not always systematic and also the fact that documentation for the DiCo [3, 6] was not always up to date. As both Papillon and DiCo

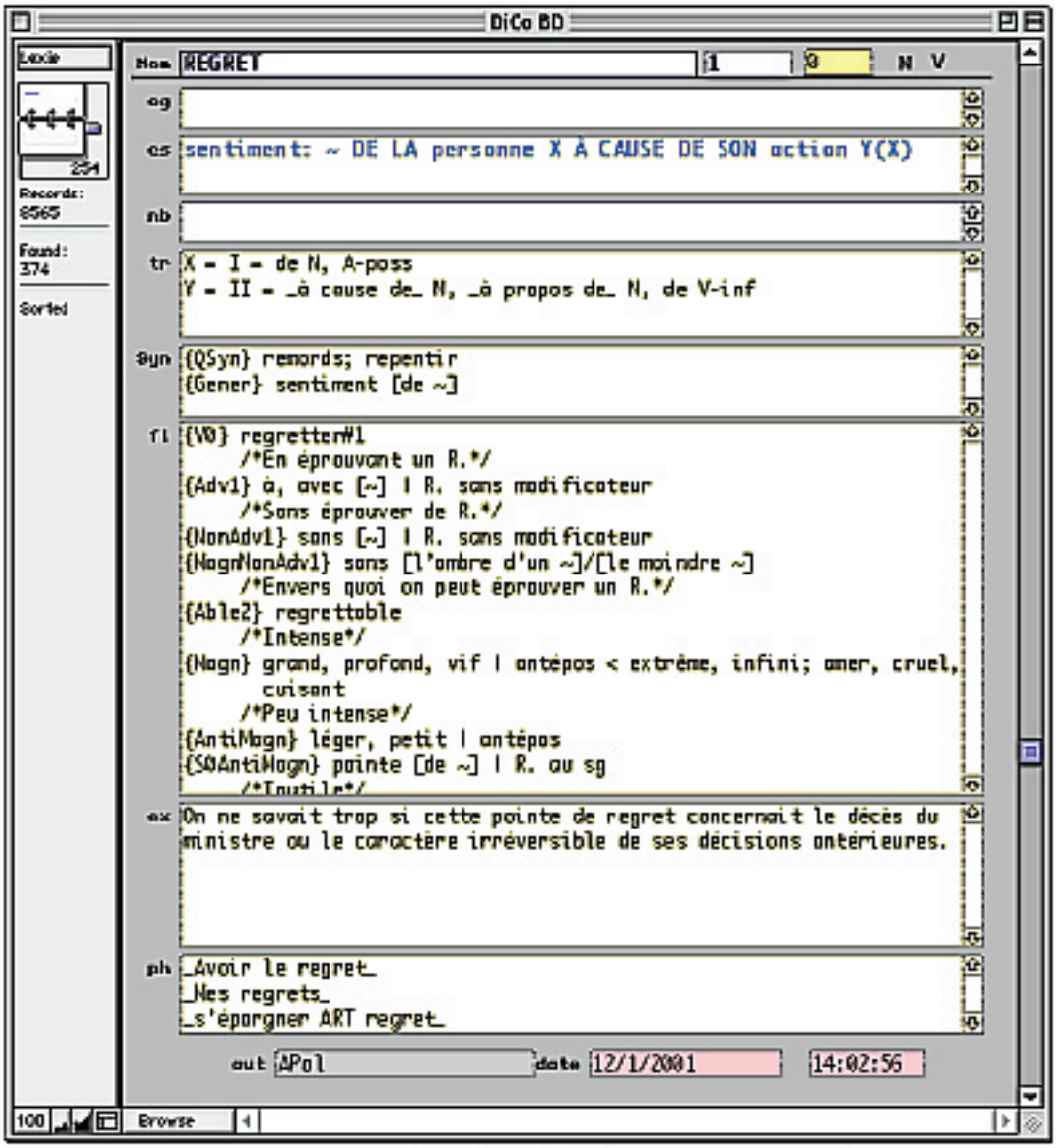

Figure 1: One DiCo entry via FileMaker Pro. This picture is taken from the master's thesis of François Lareau [3]

.

```
<lexie d:id="REGRET.1">
   <headword ln="1">regret</headword>
   <pos>n.m.</pos>
   <semantic-formula>
      <formula-label>sentiment</formula-label>
      <formula>
         ~ DE LA
         <sem-actor>
            <sem-label>personne</sem-label>
            <sem-variable>X</sem-variable>
         </sem-actor>                              [... 5 lines]
      </formula>
   </semantic-formula>
   <government-pattern>
      <mod nb="1">
         <actor>
            <sem-variable>X</sem-variable>
            <synt-variable>I</synt-variable>
            <surface-group>
               <surface>de N</surface>
               <surface>A-poss</surface>
            </surface-group>
         </actor>                                  [... 9 lines]
      </mod>
   </government-pattern>
   <lexical-functions>
      <function name="QSyn">
         <valgroup>
            <value>remords</value>
         </valgroup>
         <valgroup>
            <value>repentir</value>
         </valgroup>
      </function>                                 [... 13 lines]
      <function name="Adv1">
         <valgroup>
            <comment>En éprouvant un R.</comment>
            <value>à</value>
            <value condition="| R. sans modificateur">
               avec
               <fct-pattern>[~]</fct-pattern>
            </value>
         </valgroup>
      </function>                                 [... 99 lines]
   </lexical-functions>
     <examples>
         <example d:id="REGRET.1.e0">On ne savait trop si cette
             pointe de regret concernait le décès du ministre ou le
             caractère irréversible de ses décisions antérieures.</example>
     </examples>                                  [... 3 lines]
</lexie>
```

Figure 2: `regret.xml` generated from the tab-separated file corresponding to figure 1 . The full entry having 166 lines, for saving space in this figure, some lines have been elided and are indicated here by square brackets.

entries are strongly linked, the conversion between them cannot be limited to a simple field to field transformation and/or reordering.

Through the collaboration of Alain Polguère, Sylvain Kahane and Adil El Ghali, we obtained `dicoTab.txt` a new file of 772 entries using a similar DiCo format. From this file, we managed to create a 88 000 lines XML file with 613 lexies (337 different *words*) and 580 links between them. These entries make 14 506 references to other 7 036 entries which we would need to create in order to properly link them all... and this is not even taking into account that new references would certainly appear in these entries. These lexies are now integrated in the Papillon database.

The creation of a lexie closely follows the general structure of the XML entry shown in figure 2. As there are 8 subelements of a `lexie` element, there are eight Java classes for each of them which take as input the necessary pieces of information from the DiCo fields. We limited the use of external Java classes to the ones already used in the Papillon project: `Xerces` XML parsers and `Jakarta` regular expressions.

Here are some of the major quirks that we had to deal with when converting from DiCo to Papillon:

1. the numbering of headwords are not always consistent between both formats, so some rearrangements and even heuristics had to be used to match them;

2. in DiCo, some common informations to lexies are kept in an entry numbered 0 while in Papillon this information has to be repeated in each one;

3. part-of-speech tags do not follow the same conventions in DiCo and Papillon so some non-trivial look-up procedures had to be devised in order to make them correspond; both projects should perhaps cooperate on this conceptually simple issue;

4. *Semantic Formula* processing transforms a DiCo entry such as:

   ```
   sentiment: ~ DE LA personne X À CAUSE DE SON action Y(X)
   ```

   into a Papillon entry as follows:

   ```
   <semantic-formula>
       <formula-label>sentiment</formula-label>
       <formula>~ DE LA <sem-actor>
               <sem-label>personne</sem-label>
               <sem-variable>X</sem-variable>
           </sem-actor>À CAUSE DE SON <sem-actor>
               <sem-label>action</sem-label>
               <sem-variable dep="X">Y</sem-variable>
           </sem-actor>
       </formula>
   </semantic-formula>
   ```

This is achieved by a mix of regular expressions matching on parts of the original string in order to retrieve the formula label and content with the explaining text and the semantic actors (both label and variables).

5. *Government Pattern* processing transforms a DiCo entry such as:

```
X = I = de N, A-poss
Y = II = _à cause de_ N, _à propos de_ N, de V-inf
```

into the following Papillon one:

```
<government-pattern>
    <mod nb="1">
        <actor>
            <sem-variable>X</sem-variable>
            <synt-variable>I</synt-variable>
            <surface-group>
                 <surface>de N</surface>
                 <surface> A-poss</surface>
            </surface-group>
        </actor>
        <actor>
            <sem-variable>Y</sem-variable>
            <synt-variable>II</synt-variable>
            <surface-group>
                <surface>À cause de_ N</surface>
                <surface>À propos de_ N</surface>
                <surface> de V-inf</surface>
            </surface-group>
        </actor>
    </mod>
</government-pattern>
```

Government patterns are amongst the least systematic fields of the DiCo and some tricks and heuristics had to be used to parse them. Informations in this field are relatively tricky to enter without any strong guidelines or constraints in a text-only editor. So in a way, it is quite surprising that DiCo entries are even parsable. In particular, end of lines which should appear only at the end of patterns often occur within them for formatting purposes; moreover, some patterns are explicitly tagged with `Reg.` or `Mod.` while others are not. We thus had to remove all end of lines before separating the patterns using other hints and heuristics before applying regular expressions to separate the different parts of a pattern.

6. Lexical functions being a major component of both DiCo and Papillon, their processing is relatively involved. For example, from a DiCo entry such as

```
{V0} regretter#1
     /*En éprouvant un R.*/
{Adv1} à, avec [~] | R. sans modificateur
```

one should produce a Papillon one like the following:

```
<lexical-functions>
    <function name="V0">
        <valgroup>
            <value>regretter#1</value>
        </valgroup>
    </function>
    <function name="Adv1">
        <valgroup>
            <comment>En éprouvant un R.</comment>
            <value>À </value>
            <value condition="| R. sans modificateur">
               avec<fct-pattern>[~]</fct-pattern>
            </value>
        </valgroup>
    </function>
</lexical-functions>
```

Again in DiCo, there is a mix of new lines used as either separators between functions or for formatting purposes; although by visual inspection of the entries, it is simple to distinguish both uses, it becomes quite tricky to sort these out by program. Another annoying problem is the fact that some characters, such as commas, which act as value separators can also be used in some contexts such as examples, comments or conditions. A mix of specialized tokenizers, regular expressions and local grammars are used to separate the different uses.

7. Finally, we add some information at the end of a Papillon entry to indicate that it was generated automatically from a DiCo entry.

4 Linking lexies

The main difference between on-line dictionaries such as Papillon and DiCo and paper ones is the fact that their entries are strongly linked in order to ease their non sequential search and browsing. These links are a great asset and should be preserved when converting from one format to another. We limited ourselves to the processing of links between values of lexical

functions with the corresponding lexie, when it is defined. This processing is performed after creation of the lexies by creating a table of all headwords with the corresponding XML elements; values of lexical functions are then scanned in order to see if they refer to a lexie present in the dictionary. If so, a link is created between the two lexies; for example, `<value>regretter#1</value>` in the previous example will be replaced by

```
<reflexie xlink:href="#REGRETTER.1">regretter#1</reflexie>
```

If a corresponding value is not found, then this entry is kept in a table of undefined references which is printed at the end of this step. This is very useful for validating and checking the internal coherence of the dictionary because it makes all undefined references stand out. This list could also be used for determining the most frequently referenced entries which perhaps should be defined next. Matching headwords and lexical function values is currently done by matching the character strings, sometimes using also numbers and some information about their part-of-speech. A better coordination between DiCo and Papillon would help here also to simplify this matching process. The links could also be corrected using the GUI of Papillon.

Some other minor modifications were also done on the Papillon Schemas such as adding allowed values in some enumerations or attributes to keep informations present in DiCo but not in Papillon. This means that the basic schemas of Papillon are sound and appropriate for wide variety of uses. We detected some minor errors in the version of the DiCo we used as input.

5 Related work

Given the fact that this transformation was a "one shot deal", we might wonder if it was really worth the trouble. Currently, almost all Papillon entries have been converted from other sources; the entries are partial ones (some have only their head) and so it is important to understand the different types of conversion needed in almost all cases. The conversion process was helped by the formal definition of Papillon XML Schemas which were used as guides for programming. We think that almost all conversions will follow the overall process defined in this paper: build an internal structure of an entry starting from the input format and format the output according to the XML Schema; in our case, the use of the XML Java API was a great help because we could concentrate on creating the lexies and linking them within a DOM object which is serialized at the end of the conversion process. This opens up new ways of building Papillon entries outside of typing XML text (with or without a special XML editor or stylesheet). For many users, it will be easier to use the DiCo or their favorite dictionary editors for entering their lexies which can then transfered automatically into Papillon.

Nguyen [1] has well described the issues around dictionary *recovery* in order to build lexical resources; he defined a notion of structural and internal complexity of a dictionary and its entries. He implemented a Common Lisp system for parsing electronic dictionary

entries [1] into a special purpose internal form. Alternative dictionaries can be built from this representation using another formalism he has specified. In our case, the internal structure in a standard DOM structure which can be serialized into a XML file that can be validated and used as input for the Papillon database. Other output forms could be obtained by using standard XSLT transformations. Nguyen has managed to process more than 30 resources (1,6 million entries), one of them being 100 entries from the MSWord file of Volume 2 of the DEC [5]. He encountered the same kinds of problems we described earlier: the authors of the DEC use a mixture of formatting conventions, unfortunately not always systematic, for indicating and separating fields and features. Nguyen makes only a passing reference to the important problem of linking lexies which proved to be an interesting challenge and the gist of the information in a structured dictionary such as Papillon.

6 Conclusion

We have described our experiment in transforming DiCo entries into Papillon in one batch. We had initially thought that this would be a straightforward task but it became quite involved and it raised some interesting issues for both projects which share some important aspects as they both rely on MTT and they both make heavy use of links between entries.

This exercise was very instructive to us and productive for Papillon as it enabled the creation of about 700 XML valid entries. It is thus a good base for creating other entries. Now it can be said that Papillon has been *primed* for French and we can pursue our initial goal of creating an example generator of Papillon.

References

[1] Haï Doan-Nguyen. *Techniques génériques d'accumulation d'ensembles lexicaux structurés à partir de ressources dictionnairiques informatisées multilingues hétérogènes.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, Dec 1998.

[2] Guy Lapalme. Création d'entrées Papillon à partir du DiCo. Technical report, GETA-IMAG, Grenoble, Dec 2002.

[3] Francois Lareau. La synthèse automatique de paraphrases comme outil de vérification des dictionnaires et grammaires de type sens-texte. Master's thesis, Département de linguistique et de traduction, Université de Montréal, 2002.

[4] Mathieu Mangeot-Lerebours. *Environnements centralisés et distribués pour lexicographes et lexicologues en contexte multilingue.* PhD thesis, Université Joseph-Fourier, Sept 2001.

[1] which unfortunately often must be preprocessed with word processor macros in order to be more systematic

[5] Igor Mel'čuk *et al. Dictionnaire explicatif et combinatoire du francais contemporain: Recherches lexico-sémantiques I, II, III, IV.* Presses de l'Université de Montréal, Montréal, 1984, 1988, 1992, 1999.

[6] Alain Polguère. Towards a theoretically-motivated general public dictionary of semantic derivations and collocations for french. In *Proceedings of Euralex'2000*, pages 517–527, Stuttgart, 2000.

[7] Gilles Sérasset and Mathieu Mangeot-Lerebours. Papillon lexical database project: Monolingual dictionaries and interlingual links. In *Proc. NLPRS'2001 The 6th Natural Language Processing Pacific Rim Symposium, Hitotsubashi Memorial Hall, National Center of Sciences*, pages 119–125, Tokyo, Japon, 2001.