Eduardo Torres Schumann, Uwe Mönnich, Klaus U. Schulz

Integration Languages for Data-Driven Approaches to Ontology Population and Maintenance

Populating an ontology with a vast amount of data and ensuring the quality of the integration process by means of human supervision seem to be mutually exclusive goals that nevertheless arise as requirements when building practical applications. In our case, we were confronted with the practical problem of populating the EFGT Net, a large-scale ontology that enables thematic reasoning in different NLP applications, out of already existing and partly very large data sources, but on condition of not putting the quality of the resource at risk. We present here our particular solution to this problem, which combines, in a single tool, on one hand an integration language capable of generating new entries for the ontology out of structured data with, on the other hand, a visualization of conflicting generated entries with online ontology editing facilities. This approach appears to enable efficient human supervision of the population process in an interactive way and to be also useful for maintenance tasks.

1 Introduction

Ontologies play a key role in the Semantic Web and Knowledge Management research as a way to model the domain of application and in order to achieve an integrated access to heterogeneous data sources. Building an ontology has also become usual when developing resources for Natural Language Processing (NLP) applications, due to the need of representing meaning traditionally compiled in lexica or thesauri in a form suitable for manipulation by the computer. Although there are some differences between ontologies used in the different scenarios – e.g. some ontologies used in NLP applications are referred to as *light weight ontologies* because their lack of a specification in a formal ontology language –, they all share the fact that the process of ontology development requires considerable human effort. Moreover, ontologies have to be refined and maintained regularly in an iterative, data-driven process called the ontology learning cycle (Maedche and Staab, 2001). This tasks constitute a determining factor in the development of NLP applications and have also been identified as the "bottleneck on the way to the semantic web". If a large amount of existing data is intended to be integrated through ontologies, approaches need to work in a data-driven, automatic way but at the same time enable efficient human supervision.

The work we report on here is carried out in the framework of a larger project that aims to encode vast encyclopedic and common purpose knowledge deposited in a knowledge base called the EFGT Net, which is intended to be used for different NLP applications like semantic annotation and thematic reasoning. The EFGT Net is managed by means of a formal language been specially designed for this purpose. Rather than providing an exhaustive, precise definition of the formal meaning of a concept, the aim of this language is to place concepts in the thematic space represented by the EFGT Net. One main advantage of the EFGT Net language it can easily handle nets with more than 10^5 concepts. Further details on motivations, design guidelines and scientific ideas behind the EFGT Net are provided by Schulz and Weigel (2003).

As one branch of development, we wished to populate the EFGT Net with large amounts of already available data, such as legacy data from previous projects, public available data like GeoNames (2007) and data automatically extracted from document resources like the Wikipedia (Wikipedia, 2007). These data sources were given in different formats, such as tables and XML. The translation of data into entries of the net turned out to be a non-trivial and interesting task. Two questions became central:

1. How to specify a computable mapping from data of distinct formats to possible concepts of the ontology?
2. How to support the user in the task of deciding which of the generated concept candidates harmonize with the resource and should be incorporated to the ontology?

Here, we introduce our technical solution to these problems, the so-called Upload Tool. As to the first question, our approach allows the user to define templates of a specific form. In the simplest case, a template can be seen as a formal definition of a new concept, expressed in an extension of the EFGT Net language with the help of variables. The template specifies how the sequence of template variables is mapped to some tuple of concept names or attributes found in the textual input data. From each image tuple found in the data, a new instantiation of the template is obtained, which gives rise to a new concept definition which can be added to the ontology. More complex templates simultaneously define several new concepts from one tuple of data. Variants of the template syntax address the problem of defining data tuples and mappings for input data coming in different formats (tables, XML). This kind of extension of the ontology specification language for template definition and data integration is what we call here an *integration language*.

In many cases, some of the "new" concepts that are obtained from template instantiation will be already defined in the ontology. Hence, to avoid inconsistencies, data integration needs some form of manual control. Note that both the amount of data to be integrated as well as the number of

concepts in the resource may be very large. As a consequence, methods and interfaces that facilitate efficient human supervision of the population process are essential when providing an appropriate answer to the second question. In our approach, candidate entries are aligned with the ontology, calculating existing entries that seem close to the candidate concept, either from a logical or linguistic point of view. A visualization of the alignment results with integrated ontology editing facilities allow the user to handle conflicts on-the-fly. This kind of immediate visual feedback about the state of the ontology with respect to certain data is also useful for tracking changes in the modeling of entries and other maintenance tasks.

An implementation of the Upload Tool has been applied for substantially extending a core ontology. By using already existing lists and data extracted specially for this purpose, we were able to incorporate to the net some 10^4 new concepts representing common named entities, most of them geographical names and names of famous people.

The paper is structured as follows. Section 2 covers related work. Section 3 introduces the formal language for concept definition in the EFGT Net, which is extended in Sec. 4 to a language for defining templates. Section 5 explains how to instantiate templates given suitable tables or XML data. Possible conflicts arising during alignment of generated concept definitions are described in Sec. 6. The Upload Tool and interactive support mechanisms are described in Sec. 7. Section 8 discusses on future work and other possible applications for templates.

2 Related Work

Obviously, the kind of integration language proposed here has many commonalities with data transformation languages such as XSLT (XSLT, 2006), since we use it to transform data from different formats into entries of the EFGT Net. Some distinctions should be made though. While transformation languages are designed to bring easily some specific data model or format into other formats, as e.g. XSLT is designed for transforming XML, our integration language is designed for the converse goal, i.e. bringing syntactically heterogeneous data into a single target model. As explained later, this is reflected in the syntax of our integration language, which has an "intensional" part based on the EFGT Net language used for constructing and querying the EFGT Net and an "extensional" part for extracting data given in different formats. The intensional part our integration language is comparable with known ontology query languages such as OWL-QL (Fikes et al., 2003) for querying ontologies stated in OWL or SPARQL (Prud'hommeaux and Seaborne, 2006) for RDF ontologies, which can also be regarded as extensions of the corresponding ontology specification format. In fact, in our approach the ontology is also queried during alignment in order to

determine already existing concepts with the same logical representation as the generated entries. Since the logical representations generated by our templates are (almost) fully specified, the use of a full-fledged ontology query language like the ones mentioned above is unnecessary.

A way of obtaining structured data from heterogeneous, semi-structured data sources like web pages consists in defining wrappers (Laender et al., 2002). The extensional part of our integration language has an analogous function to the many existing specialized wrapper definition languages (WebL, 2005; Huck et al., 1998) and is used for specifying how to extract tuples from specific data files.

In work focusing on learning and populating ontologies from text (Buitelaar et al. (2005) gives an overview), integrated frameworks a have been described (Maedche and Volz, 2001) where the ontology engineer can edit the ontology and perform other maintenance tasks as she inspects the results from the learning component. This constitues a practical solution, as learning ontologies from text remains a difficult task and automatically produced results cannot be always simply adopted. In constrast, we don't learn new concepts but generate them out a given data set, sharing with the mentioned tools the idea of assisting the user during the population process.

Sophisticated interactive methods for user guidance can also be found in work on methods for merging already existing ontologies. Although fully automatic approaches to this problem exist, see e.g. Ehrig et al. (2005), other approaches bank on human supervision of the integration process, such as PROMPT (Noy and Musen, 2000) and OntoMap (Maier et al., 2003). They share the idea of identifying anchors, i.e., corresponding concepts in the target ontologies, and the merge the structure of both ontologies basing on heuristics. Unclear cases together with possible pertinent merging operations are are displayed and the user has to take a decision for the next merging step. Anchors are manually stated (OntoMap) or identified automatically based on linguistical similarity (PROMPT). Although we also exploit linguistical simliarity during the alignment of the generated entries, the integration approach presented here is fundamentally different because, instead of heuristically merging ontology structures, we map structural relationships encoded in data files to ontological relationships.

3 The EFGT Net Language - An Overview

The EFGT Net formalism is presented in this section on an informal basis. We focus on its logical language, which will be extended to an integration language in the next sections. Concepts are captured in a EFGT Net by creating an entry consisting in

1. a unique identifier for the concept, its *ID String*, that determines the position of the concept in the net, and

2. *a set of attributes* holding a linguistical representation of the concept including the name of the concept in at least one of the supported languages, a list of synonyms, orthographical variants and flection forms, etc. as well as other data like a related URL, an associated period of time, etc.

For a set of entries, a sound underlying deduction calculus determines a directed acyclic graph (DAG) for the corresponding set of ID String, where nodes in the DAG represent the concepts and edges binary relations between concepts. Schulz and Weigel (2003) provide more details.

Figure 1 summarizes the syntax of ID Strings. Starting from the root element `()`, there are two operation for creating complex ID Strings for a new concept out of a given one. The first operation, a *local introduction*,

$$\begin{array}{lcl} \mathbb{IDS}\mathrm{tr} & := & \texttt{()} \mid \texttt{(}\,\mathbb{T}\mathrm{ype}\;\mathbb{IDS}\mathrm{tr}\;\texttt{.}\;\mathbb{N}\mathrm{um}\,\texttt{)} \mid \texttt{(}\,\mathbb{IDS}\mathrm{tr}\;\texttt{\&}\;\mathbb{IDS}\mathrm{tr}\,\texttt{)} \\ \mathbb{T}\mathrm{ype} & := & \texttt{(e|E|F|g|G|t|T)} \\ \mathbb{N}\mathrm{um} & := & \mathbb{D}\,\texttt{(o|}\,\mathbb{D}\,\texttt{)}^{*} \\ \mathbb{D} & := & \texttt{(1|...|9)} \end{array}$$

Figure 1: ID String syntax.

is used when the new concept can be sufficiently characterized as a set or a set element that narrows the meaning of a more general concept. For example, the concept "Cities" can be regarded as a set of geographical locations (different cities) narrowing the more general concept "Locations", and "Oslo" could be an element of the set "Cities". This is represented in the formal language by marking the ID String of the refined concept with a special type which corresponds to the kind of specialization (i.e. the "kind" of set or set element), as well as an index for enumeration. In the example, "Locations" could be coded as a refinement of the top node `()` by marking it with the type `G` of sets of geographical entities and the index `1`, i.e. `(G().1)`. Analogously, "Cities" could be coded as `(G(G().1).1)` and "Oslo" as the (third) element of "Cities" by using type `g` for geographical elements, `(g(G(G().1).1).3)`. The complete set of types, which motivates the name EFGT Net, contains:

`E,e` Type `E` denotes a set of *E*ntities like `composers` whereas type `e` denotes a singleton entity like *J. S. Bach*.

`F` Type `F` denotes a thematic *F*ield like *quantum physics*. Since every thematic field can be regarded as a set of subfields, there is no type `f`.

`G,g` As mentioned before, type `G` denotes *G*eographical sets like *rivers* whereas type `g` stands for singleton geographic instances like *the Alps*.

`T,t` Finally, type `T` denotes a set of *T*emporal periods like *epochs in art*, and type `t` denotes an individual temporal interval like *September 11th 2001*.

The other way to create a new ID String is by combining two ID Strings with the operator `&` to form a new one. This is called a *concept intersection*. If the component ID Strings are sets of the same type, the new concept stands for the intersection of the sets. When combining a set with an individual, the new ID String represents a subset where the individual acts as a modifier. In the remaining cases, the intersection represents a new thematic field, the exact meaning is left open. E.g. combining "Persons", `(E().1)`, and "Science", `(F(F().1).2)`, would yield "Persons in Science/ Scientists"), `((E().1)&(F(F().1).2))`. "European Countries" may result from joining the identifier for "Europe", `(g(...).1)`, and "Country", `(G(...).2)`, to `((g(...).1)&(G(...).2))`.

The attributes used in addition to the ID String for characterizing the concept are qualified by means of a *semantic type* indicating which kind of information it holds (birth date of a person, the number of inhabitants of a city, a company's web-page, etc.). Additionally, a *syntactic type* specifies how the information is conveyed (as a date, proper name, URL, etc.) and a *language type* the language used. The list of semantic and syntactic types is open and can be extended to accommodate different requirements.

4 From the EFGT Net Language to an Integration Language

In this section we start developing our particular integration language for the EFGT Net. We use it for stating *templates*, which specify how data can be integrated in the ontology. As a starting point, consider the very simple data example about Switzerland's geography compiled in Table 1. Suppose

Canton	District	Capital
Thurgau	Bezirk Weinfelden	Weinfelden
Thurgau	Bezirk Bischofszell	Bischofszell
Wallis	Bezirk Brig	Brig-Glis

Table 1: Geographical data about Switzerland

you want to add each district in the table to an EFGT Net already holding the cantons. For each canton, one may want to introduce a new set of type `G` for receiving the canton districts and then add each district with type `g` to the corresponding district set.

We first explain how templates without variables may be used to define new candidate entries. In our integration language, the template

$$(_{districts}\ \texttt{G}[\text{Thurgau}].\texttt{n})$$
$$districts.\texttt{name.en.name} = \text{“districts in canton Thurgau”}$$

constructs a suitable ID String for a new concept named "districts in canton Thurgau" under the concept "Thurgau" (we may later add the corresponding

districts in a second step). The first line provides the skeletal structure of the ID String for the new concept expressed in the extended ID String language. It queries the net for the concept between the square brackets, "Thurgau", takes its ID String and defines a new set of type `G` as a local introduction with a fresh index, represented by `n` in the expression. Assuming the ID String for "Thurgau" is `(g(G().2).3)` in the net, the ID String generated by this template would be `(G(g(G().2).3).1)`, where, for simplicity, it is also assumed that there are no further local introductions of type `G` under "Thurgau" and taking `n` = `1` provides a fresh index. The string '*districts*' in tiny letters, a *concept anchor*, marks the expression between the parentheses as the ID String for a new concept and is used to assign an attribute representation to the generated ID String. Here fore, the second line of the template specifies the name of the concept as "districts in canton Thurgau", where `name` is (incidentally) both the semantic and syntactic type and `en` the language type. The entry set generated by this template is then {(`(G(g(G().2).3).1)` , {`name.en.name` = "districts in canton Thurgau"})}, which contains a single entry of the form (*ID String*, *attribute set*). In the general situation, each anchor in the template generates an entry that can be aligned with the ontology, so only templates with at least one concept anchor with its corresponding attribute specification are allowed. One template produces as many ontology entries as anchors are introduced in the template.

The real power of templates comes from the use of variables. Figure 2 shows an elaboration of the simple template above. The local introduction

$$
\begin{array}{rl}
(_{capital}\ (_{district}\ \texttt{g}(_{districts}\ \texttt{G[\#Canton].n}).\texttt{n})\&[\text{capitals}]) & \\
\textit{districts}.\texttt{name.en.name} & = \text{"districts in canton } \texttt{\#Canton}\text{"} \\
\textit{district}.\texttt{name.en.name} & = \text{"}\texttt{\#District}\text{"} \\
\textit{capital}.\texttt{name.en.name} & = \text{"}\texttt{\#Capital}\text{"}
\end{array}
$$

Figure 2: A more elaborated entry template

of type g (inner term) shows that each *district* is encoded as an element of the corresponding district set. Each *capital* (cf. outermost term) is modeled as an intersection of its district with the concept "capitals" which already exists in the net. Instead of writing a new template for each capital, we use variables that point to values in the table. These variables are represented in Fig. 2 by the strings `#Canton`, `#District` and `#Capital`. When executing the template for the given data set, candidate entries are produced, the description using the concepts found in the table. Details of the mapping from template variables to data will be discussed in the next section. For the moment, if we assume that the template in Fig. 2 is correctly instantiated with the values found in the first row of Table 1, three candidate entries are generated, which are depicted in Fig. 3. Here we assume that there are already two districts in the district set, so "Bezirk Weinfelden" receives the

{ ((G(g(G().2).3).1), {name.en.name = “districts in canton Thurgau”}),
((g(G(g(G().2).3).1).3), {name.en.name = “Bezirk Weinfelden”}) ,
(((g(g(G(g(G().2).3).3).1).3)&(G((G().1)&(F().2)).3)),
{name.en.name = “Weinfelden”}) }

Figure 3: Entry set generated by the elaborated template for the first row in Table 1

index 3. Note that for the following instantiation, which uses the second row of Table 1, a redundant entry would be generated for “districts in canton Thurgau”. Such redundancies are handled later during the alignment of entries with ontology.

The complete extended ID String syntax for template definition is summarized in Fig. 4. Compare it with Fig. 1. The last three alternatives for $\mathbb{IDStr}'$ constitute the core of the extension, where the first of them represents

$$
\begin{array}{lll}
\mathbb{IDStr}' & := & \texttt{(}\,\texttt{)} \mid \texttt{(}\,\mathbb{T}\text{ype}\;\mathbb{IDStr}'\;\texttt{.}\;\mathbb{N}\text{um}\,\texttt{)} \mid \texttt{(}\,\mathbb{IDStr}'\;\texttt{\&}\;\mathbb{IDStr}'\,\texttt{)} \mid \\
 & & \texttt{[}\,\mathbb{Q}\text{uery}\,\texttt{]} \mid \texttt{(}\,\mathbb{A}\text{nchor}\,\mathbb{T}\text{ype}\;\texttt{[}\,\mathbb{Q}\text{uery}\,\texttt{]}\;\texttt{.n)} \mid \texttt{(}\,\mathbb{A}\text{nchor}\,\mathbb{IDStr}'\;\texttt{\&}\;\mathbb{IDStr}'\,\texttt{)} \\
\mathbb{A}\text{nchor} & := & \text{Alphanum}^{+} \\
\mathbb{Q}\text{uery} & := & \mathbb{IDStr} \mid \mathbb{L}\text{it} \\
\mathbb{L}\text{it} & := & \text{String} \mid \mathbb{R}\text{ef}
\end{array}
$$

Figure 4: The extended ID String syntax.

a concept query and the other two are variants of the local introduction and concept intersection rules that introduce concept anchors. We allow ID Strings or *literals* as queries. A literal (rule $\mathbb{L}$it) is an arbitrary string value specified in the template and interpreted as a concept name or, alternatively, a value taken from data, symbolized by $\mathbb{R}$ef for reference in the syntax. References are discussed in the next section. Names for concept anchors are alphanumeric strings (rule $\mathbb{A}$nchor).

$$
\begin{array}{lll}
\mathbb{A}\text{ttr} & := & \mathbb{A}\text{nchor}\;\texttt{.}\;\mathbb{S}\text{em}\;\texttt{.}\;\mathbb{L}\text{ang}\;\texttt{.}\;\mathbb{S}\text{yn}\;\texttt{=}\;\mathbb{L}\text{it} \\
\mathbb{S}\text{em} & := & \texttt{(}\,\texttt{name} \mid \texttt{syn} \mid \texttt{url} \mid \ldots\texttt{)} \\
\mathbb{L}\text{ang} & := & \texttt{(}\,\texttt{de} \mid \texttt{en} \mid \ldots\texttt{)} \\
\mathbb{S}\text{yn} & := & \texttt{(}\,\texttt{nom} \mid \texttt{adj} \mid \ldots\texttt{)}
\end{array}
$$

Figure 5: Attribute specification syntax.

The grammar of attribute assignments via concept anchors is summarized in Fig. 5. For each anchor, literals are assigned to attributes specified by their semantic, language and a syntactic type.

5 Referring to Data for Entry Template Instantiation

In the previous section we indicated that the variables of a template are mapped to tuples of textual data for instantiation. In general, many distinct

instantiations can be obtained from one data source. Here we explain the details of this mapping for two kinds of input data, tables and XML data. This clarifies the procedural meaning of a template in presence of data encoded in these formats.

The following observation leads us to the main idea: Once a template is instantiated by some fixed tuple, the outermost anchor generates a concept of the EFGT Net which is more specific than (a descendant of) all concepts generated by other anchors of the template. More generally, if the template contains two anchors A and B, and if B is in the syntactic scopus of A, then the concept introduced by A is more specific than the one introduced by B. E.g., in the template in Fig. 2, each concept resulting from an instantiation of the anchor *district* is more specific than (a descendant of) a concept instantiation of the reference `Canton`. The anchor *capital* represents the most specific concept for each instantiation of the template. Since for a given anchor A the set of ancestors B represented in the template is unequivocal, the ancestors marked in the template *functionally depend* on the enclosing anchor. E.g., functional dependencies in our example template constitute a chain: the *capital* functionally determines the *district*, which determines the *districts* set, which determines the canton.

As explained in further detail below, each data format also encodes functional dependencies in a different way. The main idea for variable instantiation will be to extract tuples that represent functional dependencies in the data file and use these tuples in order to instantiate the functional dependencies represented in the template. This principle constraints the data files we can process using our integration language: functional dependencies represented in templates must also be encoded as functional dependencies in the data files. In this sense we say that data files must "naturally" encode the target relationships in the ontology. In practice, we found that this was not a severe restriction, allowing to leave data in its original appearance. This way, the user can define templates interactively and try different alternatives without a need of transforming data each time.

Table Data. Tables usually have at least one key column that functionally determines the values in the other columns. E.g., in Table 1 both the `Capital` and `District` columns functionally determine the other two columns, so that a chain of dependencies can be constructed. Thus, each row provides a tuple representing functional dependencies that can be used for instantiating the template. The choice here is to let the outermost anchor range over the key column and to let the other references in the template point to adequate columns, and then to evaluate the template successively for each row of the table. We mentioned that template variables are marked by the symbol `#`. If the columns in the table are named, a reference can be specified by `#` followed by the name of the column. Columns are internally numbered,

so the user can use numbers `#1`, `#2`, etc. for reference. Figure 6 shows the template of Fig. 2 with references to Table 1, so that the candidate entry set generated for the first row is exactly the set depicted in Fig. 3.

$$
\begin{array}{rl}
(_{capital}\ (_{district}\ \texttt{g}(_{districts}\ \texttt{G[\#Canton].n).n})\&[\text{capitals}]) & \\
districts\texttt{.name.en.name} & = \text{“districts in canton } \texttt{\#Canton}\text{”} \\
district\texttt{.name.en.name} & = \text{“}\texttt{\#District}\text{”} \\
capital\texttt{.name.en.name} & = \text{“}\texttt{\#Capital}\text{”}
\end{array}
$$

Figure 6: A more elaborated entry template

Sometimes, one wants to skip some rows of the table or to apply different templates depending on the value of some fields. This is achieved by using `if-then-else` statements with conditions on references, also included in the integration language. In other cases, a field of a table may contain an enumeration of concepts. This helps to collapse many rows that only differ in one column to a single row. Our language has special devices for dealing with such variations.

XML data. Each XML document can be represented as a labeled tree. In this kind of tree, some structural relationships can be regarded as encoding functional dependencies. Most prominently, an element functionally determines its parent node. If a label a occurs exactly once in each path ending with a label b, then each b element functionally determines a unique a element among its ancestors. If an element comes with a unique textual contents, then the text functionally determines the element. Furthermore, each element node functionally determines its its attribute nodes.

Figure 7 shows two panes of two different XML files encoding the data about Switzerland from Table 1. On pane A, functional dependencies in Table 1 are represented as structural relationships that are also inherently functional in XML. The text node “Weinfelden” functionally determines its element node of type `capital`, which has an unequivocal parent (with type `district`) that has an unique attribute `name`, “Bezirk Weinfelden”. So in pane A, “Bezirk Weinfelden” functionally depends on “Weinfelden”. It is easy to see that “Thurgau” is also a function of “Bezirk Weinfelden”. Of course, there a different ways to encode functional dependencies by structural relationships in an XML representation. For the extraction of tuples that represent functional dependencies, we rely on XPath expressions (XPath, 2006). The procedure is a follows. The first step is to determine by means of an absolute XPath expression a set of nodes acting as keys. The outermost concept anchor is let to range over this key set. There is a path in the document tree from each node in the key set to the root document node. References for the ancestors of the concept anchor in the

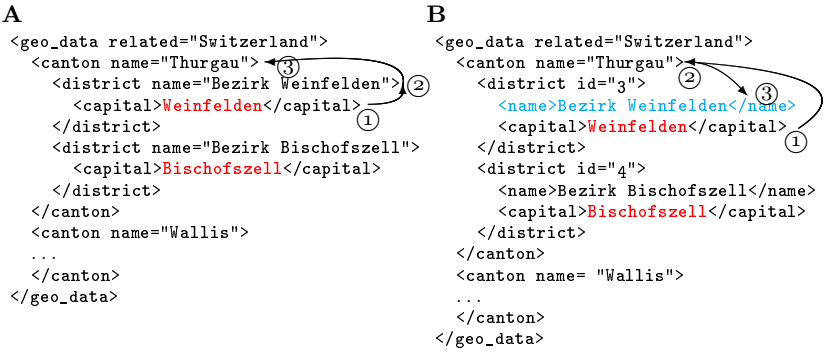

Figure 7: Two XML representations of the data in Table 1

template are defined as relative paths, interpreted on the path from the key node to the document root. In XPath terms, relative paths are evaluated against nodes in the ancestor axis of each key node or the key node itself (`ancestor-or-self` axis).

Figure 8 shows a variant of the elaborated template in Fig. 2 including XPath references that generate adequate instantiations out of pane A. The absolute path `//capital` selects the capitals of the XML document, that are identified in turn with the name of the outermost anchor concept *capital*. The other references are evaluated on each path starting at each capital node and ending on the document node. For the capital node "Weinfelden", the references `district/@name` and `canton/@name` extract "Bezirk Weinfelden" and "Thurgau" respectively from the `ancestor-or-self` axis, which is implicitly assumed and doesn't have to be stated in the template. This method is quite flexible, allowing to deal with different XML

$$
\begin{array}{rl}
(\mathit{capital}\ (\mathit{district}\ \mathtt{g}(\mathit{districts}\ \mathtt{G}[\mathtt{\#canton/@name}].\mathtt{n}).\mathtt{n})\&[\text{capitals}]) & \\
\mathit{districts}.\mathtt{name.en.name} & = \text{“districts in canton } \mathtt{\#canton/@name}\text{”} \\
\mathit{district}.\mathtt{name.en.name} & = \text{“}\mathtt{\#district/@name}\text{”} \\
\mathit{capital}.\mathtt{name.en.name} & = \text{“}\mathtt{\#//capital}\text{”}
\end{array}
$$

Figure 8: The elaborated template adapted for pane A in Fig. 7

renditions of the same data and even with those that encode functional dependencies with structural relationships that are not inherently functional in XML. E.g., pane B in Fig. 7 encodes the name of the district in an element node `name` instead as an attribute like in pane A. The relationship between the district and name element is also functional in this case, because there is only one child element of type `name`. But in the example, this is only incidentally so. This kind of structural relationship is not inherently

functional in XML, because the XML data model allows that an element has more that one element child of the same type, although, of course, that `district` element may have only on `name` element can be regulated by means of a DTD or XML Schema. We call this kind of encoding of functional relationships with not inherently functional structural relationships in XML a "pseudo-attribute". Our approach also can handle pseudo-attributes. In the template we can simply replace the reference `district/@name` by `district/name` in order to obtain the same tuples as for pane A.

6 Alignment of Generated Entries

In general, generated entries can be aligned with the ontology by examining first, whether there is another concept in the ontology with an equivalent logical characterization (*logical existence*), and second, whether the concept is already *linguistically present* in the ontology. In the case of the EFGT Net, it is enough to match the generated ID String against all ID Strings in the net to decide the logical existence. Whether a concept is linguistically present in the ontology can be decided by performing a search over the attributes holding linguistic information. The cases in Fig. 9 can then be distinguished. A generated candidate entry can be considered a *new*

Log. existent	**Ling. present**	**Case name**	**Interpretation**
no	no	*Potential new entry*	Generated new entry
yes	no	*Logical clash*	1) Complementary lex. representation 2) Logical modeling too coarse
yes	yes	*Concept match*	Entry exists already
no	yes	*Name clash*	1) Logically differing concepts with same name (homonym entries) 2) Same concept but different logical modeling

Figure 9: Alignment cases

concept to be added to the ontology when there is simply nothing indicating that it collides with another concept in the ontology. *Logical clashes* can be obliterated by merging the attribute representation of both concepts. This makes sense when the colliding concept name is just a variant not included in the linguistic representation of the existing concept. It may also be the case that the semantic analysis of two different concepts is too coarse to distinguish between them. A *concept match* is given when the generated entry is indistinguishable from another concept in the ontology. *Name clashes* also have two possible interpretations. It may occur that

two semantically different concepts share their linguistical representation (homonym entries), in which case the new candidate entry may be considered for adding it to the ontology. The converse interpretation is that an already existing concept in the ontology is modeled by the template in a different way than in the net.

7 Ontology Population and Inspection with the Upload Tool

We have developed a prototype called the *Upload Tool* that interprets EFGT Net entry templates. It has a client-server architecture, where the EFGT Net resides in a RDBS backend queried by the web client. The client interprets the template language and performs the alignment. Figure 10 shows a screen-shot of the Upload Tool. The upper part contains the entry template, submitted as a file in a previous step. The entry template can be edited and evaluated online. The results of the alignment are displayed in the lower part of the window as a list of entries for each template instantiation. When there is a concept match, entries are colored red, while potential new entries are displayed in green. Check-boxes allow for selecting green entities and uploading them to the database, where inference takes place and the structure of the net is rearranged for accommodating new concepts. Blue entries have already been considered in a previous template instantiation and don't need to be considered again.

When conflicts appear, different facilities that correspond to the different interpretations listed in Table 9 are provided to user for handling the conflict. In the case of logical clashes, a warning appears and the user can decide to run a facility for merging the attribute (lexical) representation, or, corresponding to the other interpretation, a tool for editing and refining the ID String of the conflicting concepts. For efficiently handling name clashes, there are two modes available that respectively enable or disable the creation of homonym entries. This is useful depending on the data considered. E.g., enabling homonym entries is self-evident when uploading geographical data, where homonyms are usual. Different people can also have homonym names, but when integrating names in the EFGT Net one may want to check if incoming data corresponds to a truly new person or represents a refinement in the modeling of an existing entry, so disabling homonyms is better choice in a first step. Fig. 10 shows a name clash for "Thurston Moore", where an additional window showing the parents of the already existing entry has been opened for clarification. When the name clash is interpreted as a difference in the logical modeling between the existing and the generated entry, a further tool for merging the corresponding ID Strings can be started directly from the client interface.

Posterior changes in uploaded data can be easily tracked with the Upload Tool, because once the net has been populated with a specific file, the same

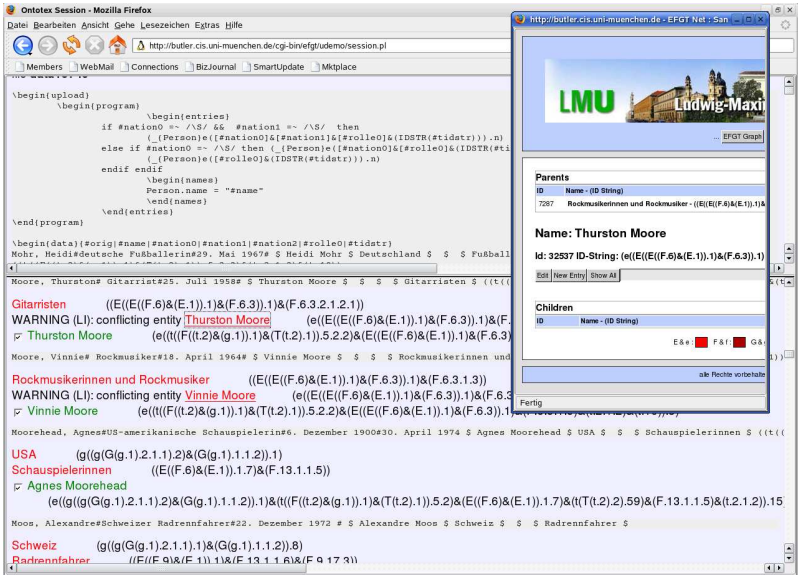

Figure 10: Alignment results in the Upload Tool

file always can be retried. If some entries have changed in the meanwhile, they will appear as name clashes when aligning them again with the original template. This can also be regarded as focused view to the ontology on the basis of some data, allowing to inspect the ontology thematically.

8 Future work

Templates are useful for maintaining and populating ontologies with pertinent data that is already available. We see additional applications for templates we want to investigate in future. Storing data together with related templates could be an easy way to create thematic ontology modules one can then combine in order to obtain customized ontologies. Maintaining a template library also could be useful for further automatizing the data integration process as well as for providing support when data acquisition and document browsing take place in an integrated scenario, as we proposed in Weigel et al. (2006).

References

Buitelaar, P., Cimiano, P., and Magnini, B., editors (2005). *Ontology Learning from Text: Methods, Evaluation and Applications*, volume 123 of *Frontiers in*

Artificial Intelligence and Applications. IOS Press.

Ehrig, M., Staab, S., and Sure, Y. (2005). Bootstrapping Ontology Alignment Methods with APFEL. In Gil, Y., Motta, E., Benjamins, V. R., and Musen, M. A., editors, *Int. Semantic Web Conference*, volume 3729 of *LNCS*, pages 186–200. Springer.

Fikes, R., Hayes, P., and Horrocks, I. (2003). OWL-QL: A Language for Deductive Query Answering on the Semantic Web. Technical Report KSL 03-14, Stanford Univ.

GeoNames (2007). GeoNames. http://www.geonames.org/.

Huck, G., Fankhauser, P., Aberer, K., and Neuhold, E. (1998). JEDI: Extracting and synthesizing information from the web. In *Proc. of COOPIS*.

Laender, A., Ribeiro-Neto, B., Silva, A., and Teixeira, J. (2002). A brief survey of web data extraction tools. In *SIGMOD Record*, volume 31.

Maedche, A. and Staab, S. (2001). Learning ontologies for the semantic web. In *Workshop on the Semantic Web (SemWeb)*.

Maedche, A. and Volz, R. (2001). The Ontology Extraction and Maintenance Framework Text-To-Onto. In *2001 IEEE Int. Conf. on Data Mining (ICDM'01). Workshop on Integrating Data Mining and Knowledge Management*.

Maier, A., Schnurr, H.-P., and Sure, Y. (2003). Ontology-based Information Integration in the Automotive Industry. In *Proc. of the 2nd Int. Semantic Web Conference (ISWC2003)*, volume 2870 of *LNCS*, pages 897–912. Springer.

Noy, N. F. and Musen, M. A. (2000). PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proc. of the National Conf. on Artificial Intelligence (AAAI)*, pages 450–455, Austin, TX.

Prud'hommeaux, E. and Seaborne, A. (2006). SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/. W3C Candidate Recommendation.

Schulz, K. U. and Weigel, F. (2003). Systematics and Architecture for a Resource Representing Knowledge about Named Entities. In *Proc. Workshop on Principles and Practice of Semantic Web Reasoning*, pages 189–207.

WebL (2005). Automating the web language. http://research.compaq.com/SRC/WebL.

Weigel, F., Schulz, K. U., Brunner, L., and Torres-Schumann, E. (2006). Integrated Document Browsing and Data Acquisition for Building Large Ontologies. In *Proc. of the 10th Int. Conf. on Knowledge-Based & Intelligent Information & Engineering Systems (KES)*.

Wikipedia (2007). Wikipedia, the free encyclopedia. http://www.wikipedia.org.

XPath (2006). XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath.

XSLT (2006). XSL Transformations (XSLT), Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xslt.