ABSTRACT
The conceptual design of the Extract – Transform – Load (ETL) processes is a crucial, burdensome, and challenging procedure that takes places at the early phases of a Data Warehouse project. Several models have been proposed for the conceptual design and representation of ETL processes, but all share two inconveniences: they require intensive human effort from the designers to create them, as well as technical knowledge from the business people to understand them. In a previous work, we have relaxed the former difficulty by working on the automation of the conceptual design leveraging Semantic Web technology. In this paper, we built upon our previous results and we tackle the second issue by investigating the application of natural language generation techniques to the ETL environment. In particular, we provide a method for the representation of a conceptual ETL design as a narrative, which is the most natural means of communication and does not require knowledge of any specific model. We discuss how linguistic techniques can be used for the establishment of a common application vocabulary. Finally, we present a flexible and customizable template-based mechanism for generating natural language representations for the ETL process requirements and operations.

Categories and Subject Descriptors
H.2.1 [Database Management]: Logical design - data models, schema and subschema.

General Terms
Algorithms, Design, Documentation.

Keywords
ETL, Data Warehouses, Conceptual Model, Natural Language, Ontologies, Semantic Web, Metadata.

1. INTRODUCTION
Data Warehouses (DW) are complex systems employed to integrate the organization’s data from several distributed, heterogeneous sources. They provide an appropriate infrastructure for efficient querying, reporting, mining, and other advanced analysis techniques. The design and the population of DW through Extract–Transform–Load (ETL) processes constitute a troublesome and time-consuming task with a significant cost in human, system, and financial resources [30]. Meanwhile, the complexity of DW environments is rising every day, and data volumes are growing at a significant pace. The DW administration and design group should manage enterprise information efficiently, and with high quality results. Approximately 30-50% of the DW design process is spent on analysis, to ensure that source systems are understood and that there is alignment between the involved parties: business and administrators. Usually, such design process involves the representation of the business requirements into a conceptual design (Fig. 1-I) and it is continuously refined through a feedback process (Fig. 1-II). An integrated metadata strategy may reduce the time needed, lower risk, and produce an ongoing record of understanding, useful in the whole DW lifecycle. Hence, it is of great importance to provide a formal, explicit and well-defined way to represent all the parameters and properties guiding this early stage. Moreover, the proposed solution should be simple and comprehensive enough to be used by all the parties involved, which usually have different knowledge background and communication codes.

Several research approaches have been proposed for the ETL conceptual design [12, 29, 30] following either ad hoc solutions or being based on well-established languages, e.g., UML. In all cases, there is the requirement that the involved parties should have good knowledge of the design method used. On the other hand, commercial ETL tools [e.g., 5, 6, 15, 16] either do not provide solutions for conceptual design at all or they use ad hoc formalism. For example, in Informatica’s PowerCenter, one may use the PowerCenter Data Stencil and the MS Visio to create conceptual designs. Additionally, in several real-world projects Excel spreadsheets are used for the task of representing the conceptual solution. Although these approaches seem quite simple to use, it is questionable how practical and effective they are, since they suffer from lack of precise metadata. In a previous work, we demonstrated that an ontology-based approach is suitable for capturing the information needed for the conceptual design of ETL processes [24]. In that setting, the burdensome task of identifying the appropriate mappings and the necessary transformations is reduced to the semantic annotation of each data store by a domain ontology; a task that is fairly simpler since each data store should be mapped to the ontology independently of the rest. Having the ontology annotating the data stores, correspondences between the sources and the data warehouse are inferred automatically by a reasoner. Although the outcome of this process (Fig. 1-I) is extremely rich in terms of semantics, it is expressed in a formal language (in OWL-DL [26]), which does not favor the communication among the different parties and makes the validation of the result harder for the designer (Fig. 1-II).
In this paper, we tackle this problem by exploiting the fact that the ontology contains all the appropriate metadata describing the ETL process and we translate the result of the reasoner into a narrative, which represents the most natural means of communication. In particular, we complement our previous effort, focusing on the establishment of a common application terminology and the use of the ontology, as a common language, to produce a textual description of meaningful reports concerning the outcomes of the ETL design phase, namely the data store annotations and the generated ETL scenario. Regarding the former, having a textual form of this information, instead of a symbolic expression in a formal language, facilitates the reading and understanding of the properties of the sources, as well as their comparison to the target requirements. Similarly, a textual description of the generated conceptual ETL operations makes it easier for the involved parties to validate the design, and to generate reports and documentation regarding the activities taking place (Fig. 1-II). For producing such reports, we devise a template-based technique, which is comprehensive, flexible, and customizable. It allows also to group together related information, so that the generated output is more concise, and thus, more comprehensive.

In summary, the main goal of our research is to provide a formal way to represent and organize automatically the outcome of the initial communications at the early phases of a DW project as a textual description that resembles natural language (NL), in order to facilitate the gathering and the validation of the appropriate metadata for the conceptual design of ETL processes.

The usefulness and importance of this goal has already been stressed in the literature. Previous work emphasizes the benefits of the automatic technical document generation. It is denoted that automatic document generation is probably best justified in terms of “guaranteeing consistency of documents with the actual designs, guaranteeing that relevant standards are followed, and simplifying the process of updating documents to react to changes in the documented hardware”.

Notice that by providing NL reporting for conceptual ETL design, we do not aim at substituting existing conceptual models; rather, we offer a complimentary functionality to them. NL reports can be used without requiring particular technical knowledge or facing practical problems; e.g., problems of visualizing ETL designs in limited screen real-estate. Additionally, recent studies suggest that the usage of NL techniques improves the quality of conceptual models and allows generation of reports in multiple languages without human intervention.

Contributions. Epigrammatically, the main contributions of our work are the following:

− We point out the usefulness of representing the formal requirements and specifications of ETL processes in a comprehensive textual format, resembling NL, to facilitate the communication among the involved parties and the overall process of design, implementation, maintenance, and documentation.
− We demonstrate that an ontology, apart from constituting a formal model for capturing application requirements and automatically inferring correspondences among the sources, can also serve as a means for verbalizing these requirements.
− We discuss how a common application terminology can be established semi-automatically, using linguistic techniques.
− We introduce a template-based technique to represent the semantics and the metadata of ETL processes as a narrative, based on information stored in the ontology, which captures business requirements, documentation, and existing schemata.
− We present mechanisms to facilitate the customization and tailoring of reports to meet diverse information needs, as well as the grouping of related information to produce more concise and comprehensive output.

Outline. The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 presents our framework for ontology-based design of ETL processes. Section 4 focuses on the use of linguistic techniques to draw a common application terminology from the source and DW schemata. Section 5 briefly describes our previous work on the generation of ETL designs. Section 6 discusses the issue of lexicalizing, and presents a template-based technique for translating the ETL requirements and operations from a formal language to a narrative. Section 7 discusses implementation issues and, then, Section 8 concludes our results.

2. RELATED WORK
This section presents efforts about the conceptual design of ETL processes, ontology creation, ontology translation, and NL applications in databases.

Conceptual models for ETL. Several approaches deal with the conceptual part of the design of an ETL scenario. A plethora of commercial ETL tools exists as well, but most of them offer limited support at the conceptual level. In general, these solutions focus mainly on the graphical design and representation of ETL processes. Each solution requires extra knowledge from the user in order to be familiarized with the symbolisms used, even if that concerns a well-known – but, only to the technical people – modeling technique, such as UML or Model Driven Architecture (MDA). Nevertheless, we do not aim at depuitizing the use of a model; rather, we enhance conceptual models with easy to understand and review reporting capabilities. In addition, a semi-automated method exploiting ontologies for the design of multidimensional DW’s is presented in [21]. Our work is complementary to that as the ontology is used for the design of the ETL process. Furthermore, that work does not deal with any reporting issues, which is our main focus in this paper.

Ontology creation. In [11], the authors give an overview of the terminology extraction methods and they show how to detect inconsistencies in the requirements document and how to extract an ontology after eliminating these inconsistencies. The issue of the results’ validation is also considered. Finally, they conclude that natural language processing is mature enough to be applied to ontology extraction in the context of requirements engineering, in spite of the necessary manual intervention. Other approaches have dealt with the issue of resolving differences in terminologies in distinct data sources.

Ontology translation. There have been efforts towards the generation of textual representations from ontologies.
These approaches constitute general-purpose ontology verbalizers, and thus, are agnostic of the types of classes, properties and operations used in our case for semantically describing the data stores and inferring correspondences among them. Hence, even though in principle it would be possible to use one of these approaches, the resulting output would be too verbose and redundant, failing to focus on the aspects of interest from the perspective of the ETL design task. It would also be more difficult to customize the output and achieve different levels of granularity according to particular information needs.

**NL and databases.** Typically, works related to database and NL research have dealt with the task of the automatic extraction of semantics from NL, which is the inverse problem of the one we are dealing with [eg., 26, 28]. More related to our context are works dealing with validation and automatic documentation. In the former category, research efforts discuss the issue of conceptual schema validation based on texts generation from the conceptual specification to NL [20], as well as the specification and validation of a conceptual schema considering the issue of detection of synonyms and homonyms using NL techniques [14]. In the latter category, the automatic generation of technical documentation and reports [e.g., 2, 19, 23] is considered.

In general, our approach is different in that it is the first, as far as we are aware of, that employs NL techniques for facilitating and clarifying the conceptual ETL design through the production of reports of the requirements and internals of ETL processes. Previous efforts have dealt with the generation of well-formed models from the analysis of (un-)structured information. (In fact, some of these results can be adapted to our case either for the construction of a global ontology from different sources or the validation of the produced reports.) The results on automatic documentation and ontology construction using NL techniques are not directly applicable to our environment, mostly due to the existence of complex and composite transformations (e.g., functions, aggregations, SK assignment, and SCD Type 1/2/3) that frequently exist in ETL processes and cannot be resolved straightforwardly using linguistic techniques. A preliminary version of this work [25] discussed the need for NL reporting and contained generic guidelines for using templates for that goal. This new effort contains the full-featured framework of our approach, the handling of terminology issues, the lexicalization process, and a detailed presentation of our template mechanism containing among others a palette of built-in functions and a set of macros for handling data store annotations and generic ETL conceptual transformations.

### 3. SYSTEM FRAMEWORK

In this section, we present our metadata-driven framework for ontology-based ETL conceptual design and reporting. The underlying idea is the use of an ontology to formally and explicitly capture the semantics of the involved data stores, and consequently guide the generation of the ETL process. Our approach comprises three main phases, as depicted in Fig. 2.

The first phase, called the **Pre-processing Phase**, aims at extracting semantics from the schemata of the available data stores. This proceeds in two steps, namely **term extraction** and **term selection**. First, for each data store schema, a set of terms is extracted, which conveys the semantics of the information contained in it. Then, these sets of terms are fused, to derive the final “terminology” or “lexicon” for the application at hand. These selected terms will be used next as the backbone for creating the ontology for the ETL design and reporting. We discuss this phase in Section 4.

The second phase, called the **ETL design phase**, is the phase where the ontology is constructed and the conceptual specification of the ETL process is generated. In particular, it involves three main steps: **ontology construction**, **data store annotation**, and **ETL generation**. Section 5 gives a brief overview of this phase.

The third phase, referred to as the **Reporting Phase**, is responsible for generating reports regarding the ETL process, as well as the involved data stores, in a format resembling natural language. This involves the lexicalization of entities used in the data store annotations and/or in the specification of the ETL process, as well as a template mechanism to generate the reports. The reporting phase is presented in Section 6.

Before proceeding with the details of our approach, we introduce a reference example for better illustrating the concepts introduced.

**Example ETL scenario.** We consider a source data store **DS_SPart**, and a target data store **DW_SupPart**, containing information about parts provided by several suppliers. Each part and supplier is identified by a unique ID. The stores contain information about the date the parts were purchased and their price. DW_SupPart keeps records only of parts purchased after the year 2000 and having from 2 to 5 suppliers. Prices in DS_SPart and DW_SupPart are recorded in Dollars and Euros, respectively. DS_SPart records the quantity of stored parts for each individual storage location, while DW_SupPart records the total quantity over all locations. Lastly, DS_SPart contains information for parts in the categories software, hardware or accessories (S/H/A), while DW_SupPart contains only software or hardware parts (S/H).

The schemata of the data stores are illustrated in Fig. 3(a), while Fig. 3(b) displays a set of corresponding terms extracted semi-automatically during the pre-processing phase. Next, the application ontology is created, represented by the graph in Fig. 4. Using this ontology, and OWL-DL constructs, the two data stores are annotated, respectively, by the following formal expressions:

```
DS_SPart = SuppliedPart ⊓ hasPartID ⊓ suppliedBy ⊓ belongsTo {software, hardware, accessories} 

DW_SupPart = SuppliedPart ⊓ hasPartID ⊓ suppliedBy ⊓ purchasedAtYear.LaterThan2000 ⊓ hasPrice.Dollars ⊓ hasPrice.Euros ⊓ hasQuantity.TotalQuantity
```

The reasoner then identifies the appropriate conceptual operations that should take place. Table 1 contains the list of transformations identified for this example. Note that due to space consideration,
The design process is complemented by our reporting functionality. Different kinds of reports can be produced covering the internals of the ETL process. An example textual description corresponding only to the information about price follows:

The DS_SPart data store has price of type dollars. The DW_SupPart data store has price of type euros. The transformation from DS_SPart to DW_SupPart data stores requires the following operations: retrieve, convert, and store. In more detail:

- retrieve price from SPart.value
- convert price from dollars to euros
- store price to SupPart.price

### 4. PRE-PROCESSING PHASE

This phase deals with the different terminologies of the data stores, aiming at establishing a common vocabulary among them, in order to overcome naming conflicts. This is performed in two steps. In the first step, the names of the elements of each individual data store are analyzed and mapped to concepts using a (domain specific, if available) thesaurus. This is accomplished by combining string matching techniques, such as the edit distance, with structure-based schema matching techniques to exploit the structure of the elements [17]. This process is semi-automatic, meaning that the designer has to verify the extracted terms and provide the correct term for elements that were mapped wrongly or not mapped at all. The amount of manual effort required depends on the quality of the naming scheme that was used during the initial design of each particular data store.

In the second step, the concepts identified for each individual data store are merged to create a unified terminology, which will be used as the basis of the constructed application ontology. The merging process is based on identifying synonyms (e.g., price, value, cost) and hypernyms (e.g., address, location) among the terms. When two or more synonyms are identified, they are replaced by a single term. By default, the preferred term to be used as representative for the list of synonyms is the one that was extracted from the target schema, so that the derived terminology is biased towards the target terminology. When a term is found to be a hypernym of another term, it is replaced by that term. This is because the data store elements are annotated by means of classes that are defined as specializations (i.e., subclasses) of the primitive classes in the ontology (see Section 5). When a term is replaced by a synonym or hypernym, the underlying mappings, i.e., the data store elements that were related to this term, are updated accordingly. However, a log is kept, so that the mapping can be reset if the designer rejects the new mapping. Typically, this step requires less manual effort than the first.

In addition, during the design of a data store schema, it is a common practice to include a short textual description for each element in NL format. Whenever such a description is available, it is exploited in constructing the application terminology, significantly increasing the accuracy of the process, and consequently further reducing the required manual effort. Specifically, during the first step, the presence of a textual description of the element can guide the identification of the corresponding term, given that typically the appropriate term to be associated to that element is included in that description. In the second step, a comparison of the elements’ descriptions, when available, helps in identifying synonyms, as well as distinguishing between homonyms.

### 5. ETL DESIGN PHASE

During this phase, an ontology is constructed, using as a basis the previously extracted terminology, and exploiting also available knowledge and requirements regarding the data stores. This ontology is then used to annotate the data stores and drive the specification of the ETL process. A detailed analysis of this task has been presented in a previous work [24]. Here, we present only the basic ideas, and the corresponding terminology and notation, which are necessary for describing the reporting mechanism in the next section. More specifically, the ontology comprises:
Table 2. Generic types of frequent conceptual ETL operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRIEVE</td>
<td>Retrieves tuples from a source element</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>Extracts parts of tuples from a source element</td>
</tr>
<tr>
<td>MERGE</td>
<td>Merges tuples from several source elements</td>
</tr>
<tr>
<td>JOIN</td>
<td>Joins tuples from several source elements</td>
</tr>
<tr>
<td>RANGE_FILTER</td>
<td>Selects tuples based on a specified value interval</td>
</tr>
<tr>
<td>VALUE_FILTER</td>
<td>Selects tuples based on a specified value set</td>
</tr>
<tr>
<td>CARD_FILTER</td>
<td>Selects tuples based on attribute cardinality</td>
</tr>
<tr>
<td>CONVERT</td>
<td>Converts between different template types</td>
</tr>
<tr>
<td>AGGREGATE</td>
<td>Aggregates the values of an attribute</td>
</tr>
<tr>
<td>DUPL</td>
<td>Removes duplicate tuples</td>
</tr>
<tr>
<td>STORE</td>
<td>Stores tuples to a target element</td>
</tr>
</tbody>
</table>

- a set of classes, $C_0$, representing the concepts of interest in the application domain;
- a set of properties, $P_0$, representing attributes of these concepts or relationships between them;
- a set of classes, $C_{TP}$, denoting the ranges of properties in $P_0$;
- three sets of classes, $C_{TP}$, $C_{CR}$, and $C_{DB}$, representing, respectively, different representation formats or units of measurements for the values of an attribute (e.g., different currencies), value intervals, and enumerations of distinct values;
- a property, `convertsTo`, relating classes in $C_{TP}$, indicating that a conversion from one format to another is possible;
- two sets of classes, $C_0$ and $C_{DB}$, denoting, respectively, aggregation functions (e.g., average, sum, count, max) and attribute values resulting from aggregation operations (e.g., average age, total cost);
- two properties, `aggregates` and `groups`, indicating, respectively, which aggregation function is used in a particular operation and which attributes are grouped.

The ontology is represented as a graph, where classes correspond to nodes and properties correspond to edges. The data stores are annotated by attaching to them logical expressions formed using the classes and properties in the ontology. Essentially, these expressions are restrictions that can be either value restrictions or cardinality restrictions. The former are denoted by $\forall P \in C$, and specify that the range of the property $P$ is restricted by the class $C$. The latter are formed as $\forall n \in P \leq n P$ or $\exists n \in P$, and specify that the cardinality of the property $P$ is exactly, at least, or at most $n$.

A reasoner is then employed to infer semantic correspondences and conflicts among the data stores and to propose a set of generic conceptual operations for transforming and cleansing source data to populate the DW data stores. We consider a set of generic conceptual operations that can capture a wide variety of transformations typically occurring in ETL processes, as shown in Table 2.

6. REPORTING PHASE

This section discusses the components of the reporting phase, namely entity lexicalization, template creation, and report production (see Fig. 2).

6.1 Lexicalization

One important step of a typical natural language generation process is entity lexicalization, which refers to deciding which words or phrases to use for transforming an underlying message into a readable text. In our case, this issue arises when deciding which word or phrase to associate to an entity, such as a class or a property in the ontology. Note that in general, this task differs from the pre-processing phase, since the common terminology of this phase: (a) may not be ready for translation due to programming restrictions (e.g., no spaces are allowed in the names) and (b) is not necessary used for the construction of a new ontology, but for the matching with an already existing ontology that may or may not be appropriately annotated for the translation process.

Therefore, we should attach to the ontology elements appropriate labels for rendering them in a textual format. In OWL it is possible, via the `rdfs:label` property, to associate a label to an entity, which provides a human-readable version of the entity’s name. Another important feature of this mechanism is that multilingual labels are supported, so multilingual output can be generated automatically (provided that appropriate templates for the desired language are also available). However, if a label is not already provided for an entity, lexicalization is performed by automatically extracting a word or phrase from the entity’s name. First, the name is tokenized, based on common heuristic rules, such as capitalization, underscores, and dashes. Then a look up to a lexicon takes place, using common string matching techniques, such as the edit distance. If a domain specific lexicon is not available, a general-purpose lexicon, such as WordNet, is used.

In summary, due to the special characteristics of the ETL environment the following solutions fit in our case:

1. The ontology is already annotated and it contains the appropriate labels from a previous attempt to design the ETL workflow or from a similar application.
2. We use a naming convention based on the entities’ names; the consideration that the name of each entity is meaningful is a valid case for several real-world DW applications [33].
3. We create appropriate labels during the construction and/or maintenance of the ontology. Since, our application ontology represents database schemata, a label per each ontology node corresponding to a database entity is created by using the comment of this database entity; e.g., many commercial DBMS’s support a CREATE COMMENT statement on relations, views, attributes, and operators.
4. We use string matching techniques (e.g., edit distance and n-gram similarity) and a lexicon for annotating the ontology.
5. We use a hybrid-method based on different combinations of the above.

Both approaches (2) and (3) may be used in combination with (4). For instance, regarding the third case, it is not always straightforward to use the comment of a database entity as a label in an ontology entity; e.g., consider that many database entities, having probably different comments attached to each one of them, are mapped to the same ontology entity. In our approach, such issues are resolved with string matching techniques (i.e., edit-distance and q-gram similarity) and the use of a lexicon (i.e., WordNet).

6.2 Templates

In the following, we present our mechanism for the derivation of reports describing the outcomes of the ETL design phase, namely the data store annotations, and the generated ETL scenario. To provide a comprehensive, flexible, and easily customizable mechanism for generating textual representations, a template-based technique is followed. In [4], a template-based system is described as a natural language generating (NLG) system that maps
non-linguistic input directly (i.e., without intermediate representations) to the linguistic surface structure, and arguments are given against the perception that template-based systems are inferior compared to “real” NLG systems. In our case, templates have the additional benefit of providing a comprehensive and intuitive way to customize the produced output, without requiring that the DW designer should understand complex NL processing techniques.

**Template language.** The translation process is realized by suitable templates, which are constructed composing elements provided by our template language. This is a typical template language, supporting constructs such as variables and directives, but also extended with built-in functions and macros suited to the ETL design task. Notice that the generated text may also contain HTML tags, so that highly formatted output can be produced. Next, we describe the aforementioned constructs in more detail.

**Variables.** A template may contain variables, which are denoted by their name preceded by the symbol $. When the template is instantiated, the template engine that processes it replaces each variable with a corresponding, provided, value.

**Directives.** A set of typical directives is supported, allowing for a high degree of flexibility in specifying templates. Specifically, the directives #set, #if / #elseif / #else, and #foreach are provided to set the value of a parameter, allow conditional output and iterate through a list of objects, respectively. The standard arithmetic, logic, and comparison operators are also supported.

**Functions.** The template language supports the usual arithmetic, date, and string manipulation functions. In addition, we provide the template designer with a set of built-in functions specifically tailored for the ETL environment, as shown in Table 3.

**Macros.** Macros allow simpler templates to be reused and/or combined to define more complex ones. Thus, they significantly facilitate the creation of templates. For instance, a typical use for macros is to specify how the elements of an array should be rendered. In our work, macros are also defined for the different types of restrictions, as well as for the different types of ETL operators, to specify the textual description of these elements. The designer may customize and extend the translation mechanism, by modifying these macros or defining new ones. An example general-purpose macro that renders the elements of a list follows:

```template
Macro: LIST(L)
#set ( $size = #SIZE($L) )
#set ( $counter = 0 )
#foreach( $item in $L )
  #if ( $counter == 0 )
    #TEXT($item)
  #elif ( $counter == $size-1 )
    #TEXT($item)
  #else
    , #TEXT($item)
  #end
  #set ( $counter = $counter + 1 )
#end
```

In the result, a comma follows each list item, but the word ‘and’ comes before the last one. Observe that standard programming knowledge is enough for the macro creation. Apart from such macros that concern generic functionalities, we do provide as well default macros for translating data store annotations and conceptual ETL operations, as shown in Tables 4 and 5, respectively. Having constructed appropriate templates based on macros ensures the reusability and extensibility of the reporting mechanism.

**Template instantiation.** According to the reporting needs, an appropriate template is created, or an existing one is chosen, and a narrative is produced automatically out of it. This procedure is performed by the template engine, and it requires that the template is used in synergy either with the formal expression annotating a data store or with (a part of) the ETL specification. The template is instantiated by expanding any contained macros, evaluating any contained functions and directives, and assigning concrete values to its parameters. Thus, the template instantiation is realized in the following order:

- First, the macro definitions are replaced by their expansions.
- Then, the functions and variables appearing in #set directives and loop boundaries are calculated.
- Next, any loop that may exist is evaluated.
- Finally, the rest variables are instantiated.

Intuitively, the abovementioned order is justified as follows. Macros should be expanded first, in order to have all the necessary expressions. Then, before executing any loop, we have to evaluate its boundaries. For doing the latter, we have first to instantiate the variables appearing in loop boundaries, and setting the counters (often by evaluating a function that defines them). Next, all the loops should be expanded for producing the appropriate lists of variables, and afterwards, the rest variables are instantiated.

**Report production.** The format of a report is highly dependent on each application. In our experience and understanding, for the early phases of a DW design project, it is advisable to have reports expressing the parts of the ETL process by simple means. For that reason, we favor the presentation of the results as bullet-lists. However, the format of a report is defined by the creator of the respective template. Our mechanism is generic enough to support fairly rich representation formats. Next, we present a series of indicative templates to illustrate the use of the proposed mechanism and demonstrate its usefulness and flexibility.

**Case I.** A template for rendering the annotation D of a data store S is structured as follows:

```template
Template: PRINT_ANNOT(S, D)
#set ( $head = HEAD(D) )
#HEADER( $S, $head )
#set ( $res_list = #PARSE_ANNOT($D) )
#foreach( $res in $res_list )
  #if ( #PARSE_RES( $res ) == "EXACT_CARD" )
    #EXACT_CARD( $res )
  #elif ( #PARSE_RES( $res ) == "MIN_CARD" )
    #MIN_CARD( $res )
  #else
    // calls to macros for other types of restrictions
  #end
#end
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD(D)</td>
<td>The primitive class appearing in the annotation D</td>
</tr>
<tr>
<td>PARSE_ANNOT(D)</td>
<td>The list of restrictions appearing in the annotation D</td>
</tr>
<tr>
<td>PARSE_RES(R)</td>
<td>The type of restriction R</td>
</tr>
<tr>
<td>TEXT(X)</td>
<td>The textual description (lexicalization) of entity X</td>
</tr>
<tr>
<td>RANGE(F)</td>
<td>The class being the range of property F</td>
</tr>
<tr>
<td>INTERVAL(C)</td>
<td>The lower/upper bounds of the value interval specified by class C ∈ Cp</td>
</tr>
<tr>
<td>ENUM(C)</td>
<td>The list of the members of class C ∈ Cp</td>
</tr>
<tr>
<td>AGGR_FUNC(C)</td>
<td>The class related to C ∈ Cp via the property “aggregates”</td>
</tr>
<tr>
<td>AGGR_ATTR(C)</td>
<td>The list of classes related to C ∈ Cp via the property “groups”</td>
</tr>
<tr>
<td>PARSE_FLOW(W)</td>
<td>The list of operations constituting an ETL flow W</td>
</tr>
<tr>
<td>PARSE_OP(F)</td>
<td>The type of operation F</td>
</tr>
<tr>
<td>PARAMS(F)</td>
<td>The list of parameters of an ETL operation F</td>
</tr>
<tr>
<td>SIZE(L)</td>
<td>The size of the list L</td>
</tr>
</tbody>
</table>

Table 3. A set of built-in functions provided by our approach
Given the annotations of the source and target data stores DS_SPart and DW_SupPart, presented in of Section 3, the above template produces the reports depicted in Fig. 5(a) and (b).

Case 2. A template for generating textual representations of generic ETL operations is similarly specified, as shown below:

**Template: PRINT_ETL(W)**
Transformations from $source$ to $target$:

```java
#set ( $op_list = #PARSE_FLOW($W) )
#foreach( $op in $op_list)
  #if ( #PARSE_OP( $op ) == "EXTRACT" )
    #EXTRACT($C)
  #elseif ( #PARSE_OP( $op ) == "MERGE" )
    #MERGE($C)
  #elseif ( #PARSE_OP( $op ) == "JOIN" )
    #JOIN($C)
  #end
#end
```

Given the ETL operations for the running example illustrated in Table 1, the above template produces the report shown in Fig. 5(c).

Case 3. Apart from rendering a complete description of the ETL flow, a template can focus on information regarding specific aspects. For instance, the following template prints the total number of operations in a given flow, as well as the number of CONVERT operations.

**Template: PRINT_ETL_STATS(W)**

```java
#SIZE(#PARSE_FLOW($W))
#foreach( $op in $op_list)
  #if ( #PARSE_OP($op) == "CONVERT" )
    #SIZE($C)
  #end
#end
```

As shown from the sample report of Fig. 5(c), the resulting output may often contain repeated information. Even though this does not necessarily constitute a problem, since the output is often presented in a tabular form or as a bulleted list, in other cases a more concise representation is preferable. This issue is addressed by grouping together related pieces of information, a process that is commonly referred to as sentence aggregation [3, 18] in Natural Language Processing. In our case, aggregation can be achieved based on the following criteria: (a) grouping together restrictions of the same type, (b) grouping together restrictions on the same property, (c) grouping together operations of the same type, and (d) grouping together operations on the same attribute.

Although this can be done programmatically by specifying the appropriate conditions in the corresponding templates, to reduce coding effort and simplify the templates, we overload the built-in functions `PARSE_ANNOT(D)` and `PARSE_FLOW(W)` providing two new variations for each one: `PARSE_ANNOT(D,R)`, `PARSE_ANNOT(D,P)`, `PARSE_FLOW(W,F)`, `PARSE_FLOW(W,P)`. The first returns only restrictions of type R, while the second returns only restrictions applied on property P. Similarly, the third and the fourth return only operations of a given type F and on a specific property P, respectively.

Hence, using in addition the function `SIZE(L)` shown in Table 3, the previous template can be significantly simplified as follows.

**Template: PRINT_ETL_STATS_SHORT(W)**

This ETL flow contains a total of `#SIZE(#PARSE_FLOW($W))` operations. `#SIZE(#PARSE_FLOW($W,"CONVERT"))` of these are CONVERT operations.

Finally, other characteristic cases are, for instance, to verbalize the first n operations of the workflow or the operations concerning a specific property. The latter case is especially useful to track the transformations occurring on a specific attribute throughout the workflow. Another practical case is to list groups of order-equivalent transformations to help the administrator to design the execution order of an ETL workflow [21].

### 7. IMPLEMENTATION ISSUES

Our prototype uses the Java programming language and a collection of open source API’s and tools. WordNet (http://wordnet.princeton.edu/) is used as a semantic lexicon for identifying synonyms and hypernyms, during the extraction of the application terminology. The application ontology is represented in OWL. Management of the ontology is performed by means of the OWL API provided by Jena (http://jena.sourceforge.net/), a Java-based, open source framework for building Semantic Web applications. Reasoning tasks, such as inferring subsumption relationships between the ontology classes, are performed by Pellet (http://pellet.owldl.com/), an open-source OWL-DL reasoner. The module responsible for the generation of the NL representations utilizes Velocity (http://velocity.apache.org/) as the template engine. The

### Table 4. A set of macros for data store annotations

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER(S, C)</td>
<td>Each tuple in $S$ contains information about $C$.</td>
</tr>
<tr>
<td>EXACT_CARD(=nP)</td>
<td>$TEXT({P})$ exactly $n \text{ #TEXT({RANGE(S)})}$.</td>
</tr>
<tr>
<td>MIN_CARD(&lt;nP)</td>
<td>$TEXT({P})$ at least $n \text{ #TEXT({RANGE(S)})}$.</td>
</tr>
<tr>
<td>MAX_CARD(&gt;nP)</td>
<td>$TEXT({P})$ at most $n \text{ #TEXT({RANGE(S)})}$.</td>
</tr>
<tr>
<td>RANGE_RES(V,P,C)</td>
<td>$TEXT({P}) \text{ #TEXT({RANGE(S)}) in #INTERVAL({C})}$.</td>
</tr>
<tr>
<td>ENUM_RES(V,P,C)</td>
<td>$TEXT({P}) \text{ #TEXT({RANGE(S)}) one of #LIST({ENUM({C})})}$.</td>
</tr>
<tr>
<td>FORMAT_RES(V,P,C)</td>
<td>$TEXT({P}) \text{ #TEXT({RANGE(S)}) of type #TEXT({C})}$.</td>
</tr>
<tr>
<td>AGGR_RES(V,P,C)</td>
<td>Contains the $TEXT(#AGGR_FUNC({C})\text{ #TEXT({RANGE(S)}) per #LIST(#AGGR_ATTR({C}))}$.</td>
</tr>
</tbody>
</table>

### Table 5. A set of macros for generic ETL operations

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRIEVE(V,C)</td>
<td>Retrieve $#TEXT({C})$ from $V$.</td>
</tr>
<tr>
<td>EXTRACT(V,C)</td>
<td>Extract $#LIST({C})$ from $V$.</td>
</tr>
<tr>
<td>MERGE(V,C)</td>
<td>Compose $#TEXT({C})$ from $#LIST({V})$.</td>
</tr>
<tr>
<td>RANGE_FILTER(P,C)</td>
<td>Select $#TEXT({P}) #TEXT({RANGE(S)})$ in $#INTERVAL({C})$.</td>
</tr>
<tr>
<td>VALUE_FILTER(P,C)</td>
<td>Select $#TEXT({P}) #TEXT({RANGE(S)})$ one of: $#LIST(#ENUM({C})$.</td>
</tr>
<tr>
<td>AGGR(P,C,V)</td>
<td>Calculate $#TEXT({C}) #TEXT({RANGE(S)})$ per $#LIST({V})$.</td>
</tr>
<tr>
<td>CARD(P,min,max)</td>
<td>Select tuples having $#TEXT({P})$ at least $\text{min}$ and at most $\text{max} #TEXT({RANGE(S)})$.</td>
</tr>
<tr>
<td>DD(P)</td>
<td>Remove duplicate tuples for $#TEXT({RANGE(S)})$.</td>
</tr>
<tr>
<td>JOIN(C)</td>
<td>Join tuples from $#LIST({C})$.</td>
</tr>
<tr>
<td>STORE(C,V)</td>
<td>Store $#TEXT({C})$ to $#LIST({V})$.</td>
</tr>
</tbody>
</table>
Each tuple in DS_SPart contains information about a supplied part. It:
- has exactly 1 part id
- is supplied by at least 1 supplier
- has price of type dollars
- belongs to category one of: software, hardware, accessories

Each tuple in DW_SupPart contains information about a supplied part. It:
- has exactly 1 part id
- is supplied by at least 2 and at most 5 suppliers
- purchased at year in [2000,e]
- has price of type euros
- contains the total quantity per part id, supplier, day, month, year, price, category
- belongs to category one of: software, hardware

Transformations from DS_SPart to DW_SupPart:
- Compose part id from SPart.model, SPart.number
- Retrieve supplier from SPart.sCode
- Retrieve price from SPart.value
- Extract day, month, year from SPart.date
- Retrieve quantity from SPart.amount
- Retrieve category from SPart.category
- Select tuples supplied by at least 2 and at most 5 supplier
- Select purchase year in [2000,e]
- Convert price from dollars to euros
- Calculate total quantity per part id, supplier, day, month, year, price, category
- Select belongs to category one of: software, hardware
- Store part id to SupPart.partID
- Store supplier to SupPart.supCode
- Store day to SupPart.day
- Store month to SupPart.month
- Store year to SupPart.year
- Store price to SupPart.price
- Store quantity to SupPart.quantity
- Store category to SupPart.category

Fig. 5. Text representation of the data stores and the transformations for the reference example presented in Section 3

8. CONCLUSIONS
In this paper, we are dealing with the facilitation of the communication among the different parties – business and technical staff – involved in the design and development phases of a DW project. Our experience shows that a narrative is a natural way of communication; hence, we have focused on the representation of ETL processes as textual descriptions resembling NL. We have built upon our previous work on regarding the automation of the ET9 conceptual design with the use of an ontology. We have discussed that linguistic techniques can be used in synergy with that ontology and the proposed template mechanism, for translating the semantic annotations of the data stores and the required ETL conceptual operations, from a formal syntax to a textual description. As a future work, we plan to investigate advanced NLG techniques to improve the generated textual reports. One of the next steps is the deployment of our prototype in real-world scenarios.

9. REFERENCES
5. IBM. IBM WebSphere DataStage. URL: http://www-306.ibm.com/software/data/integration/datastage/