

Information Gathering in a Dynamic World

Thomas Hornung, Kai Simon, and Georg Lausen

Institute of Computer Science, Albert-Ludwigs University Freiburg, Germany
{hornungt, ksimon, lausen}@informatik.uni-freiburg.de

Abstract. Web resources with constantly fluctuating content, such as virtual market places, are becoming more and more relevant as information resources. Classic search engines, unfortunately, crawl and index the Web in sporadic intervals and therefore rely on outdated information. In this paper we present OntoGather, a framework based on ontology-driven inferences on dynamically gathered annotated instances from the Web, which consists of two main components: the Web data-extraction and annotation system ViPER and the deductive object-oriented database system Florid.

1 Introduction

Classic search engines (e.g. Google) crawl the Web in intervals and build up indexes from vast amounts of data to be able to answer user queries within a reasonable time frame. This was (and still is) well-suited for static Web pages that remain stable for a longer period of time, but today many resources on the Web are in constant flux. Prominent examples for this are virtual market places or real-time information systems, e.g. stock-exchange price services. The underlying volatile nature of the aforementioned domains necessitates a dynamic approach to support user queries of the form *what is the cheapest price for an IXUS digital camera (at the moment)?*

To solve this problem we propose an approach that relies on dynamic integration of information sources which are accessed at query time. The core of our system is Florid [1], a deductive object-oriented database system based on F-Logic [2], which operates on top of a domain-specific background ontology. It serves as inference engine used for Web resource selection and evaluates a user query on up-to-the-minute information. In this paper we deal with information extracted from HTML pages with the aid of a wrapper tool. Web resources that can be accessed via Web Service interfaces, such as WSDL APIs or RSS feeds have not been considered yet, but can be easily integrated into our system. Wrapper tools range from semi-automatic approaches, such as LIXTO [3], to fully-automatic. In our scenario a fully-automatic approach is most suitable, because the wrapper generation and maintenance effort is negligible, which suits the requirements of our dynamic world scenario best. Therefore we use our fully-automatic extraction system ViPER [4], that is able to extract up-to-date information from arbitrary HTML pages consisting of data records, which have a similar structure.

The paper is structured as follows: In section 2 we present our extraction and integration system ViPER. Next we describe the underlying background ontology in section 3. The section 4 presents the main components by an example. Finally we conclude in section 5 and give an outlook in section 6.

2 Information Integration

Aiming at a robust, fast and extensible information system we opt for our fully-automatic wrapper extraction tool named ViPER (**V**isual **P**erception-based **E**xtraction of **R**ecords). ViPER is able to extract and discriminate with high accuracy the relevance of different repetitive Web information content with respect to the user's *visual* perception of a single Web page. After ViPER has identified the most relevant *data region* the tool generates a *pattern* (extraction rule), matching similar *data records*. These data records can usually be found in static Web catalogs as well as dynamic Web pages. Since these sites are often filled with information from back-end databases by predefined templates or server-side scripts, the extraction process can be seen as reverse engineering on the basis of materialized database views which have been published in HTML pages.

3 Resource Ontology

In the OntoGather system, we expect our information sources to be organized in a domain-specific ontology. This ontology initially contains meta-information about accessible Web resources, which can be referenced by unique resource ids resolved by ViPER into Web URLs. The ontology in principle could be given in any kind of formalism, e.g. OWL. Since we are particularly interested in answering queries, we have chosen F-Logic where we can specify the background ontology and the queries themselves in the same language. To further illustrate this point, we use the following running example:

```

top[resourceID⇒integer; name⇒string].
product :: top[price⇒float].
...
digital_camera :: product[model⇒string; resolution⇒integer].
ixus :: digital_camera[name●→"IXUS"; resourceID→{23, 42}].
canon :: digital_camera[name●→"CANON"; resourceID→{12, 23}].

```

(3.1)

Example (3.1) shows an excerpt of a simple product ontology. The first expression defines the concept `top` with the *multi-valued* method `resourceID` and the *functional* method `name` via their signatures. These signature definitions are inherited to every instance and subclass. The second expression declares `product` to be a *subclass* of `top` with the additional functional method `price`. The third expression introduces the class `digital_camera` which has `ixus` and `canon` as subclasses (expression four and five), that provide an implementation of the methods `name` and `resourceID`. The method `name` has been declared to be inheritable, therefore all instances of `ixus` and `canon` will have the result of the method `name` set to "IXUS" or "CANON", respectively.

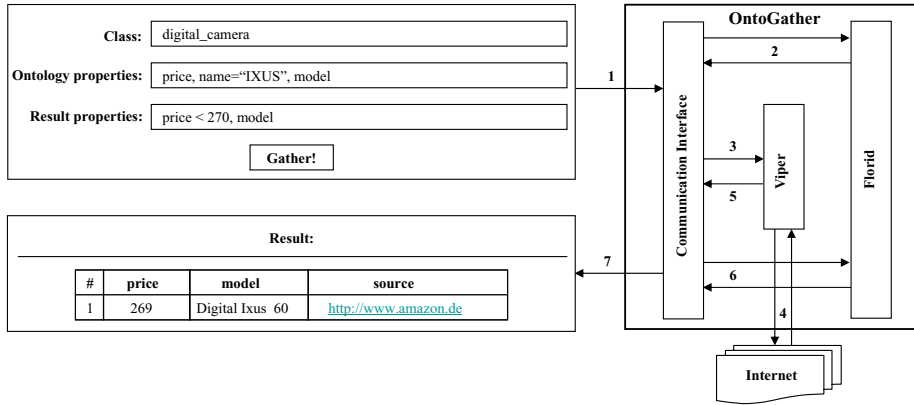


Fig. 1. OntoGather system overview

4 OntoGather by Example

Our framework is depicted in figure 1, which describes the processing of a query in seven steps. On the upper left part we see the user interface, which offers three form fields to specify a request: the field entitled "Class" narrows down the search area to a specific subpart of the ontology. The "Ontology properties" field selects the required attributes the class of interest has to provide, where it is possible to select a value (for instance `name="IXUS"`) to further restrict possible candidates. The "Result properties" field lists the desired output attributes with the constraints that have to hold. The request in figure 1 asks for products falling in the digital camera subpart of the ontology, which provide information on price, model and name, where name has to be "IXUS". Finally the result is restricted to all models having a price lower than 270. To allow most users an intuitive use of the system the interface is held simple, but support for automatic inference of remaining possible attributes and graphical selection of subparts of the graph are envisioned. Furthermore we are looking into ways to allow for more complex queries, e.g. where attributes of two distinct classes (analog and digital cameras for instance) can be requested.

After the user starts her request, the contents of the form fields are sent to the Communication Interface, that acts as a negotiation layer translating requests between ViPER and Florid (step 1). The Communication Interface generates a Florid query based on the the provided information, asking for all resources that can contribute to the answer:

$$\text{Obj} :: \text{digital_camera}[\text{price} \Rightarrow _Price; \text{model} \Rightarrow _Model; \text{name} \bullet \rightarrow "IXUS"; \text{resourceID} \Rightarrow _Resources]. \tag{4.1}$$

In query (4.1) strings starting with a capital letter are treated as variables, where strings preceded by an underscore are anonymous variables, whose bindings are not returned in the answer. The result value for the method `name` is used as restriction on the possible answer set. We forward this request to the Florid

engine which yields the following variable bindings as result (with respect to our example ontology (3.1)):

```
Obj/ixus, Resources/23
Obj/ixus, Resources/42
```

(4.2)

thus completing step 2. The variable bindings are processed by the Communication Interface and passed to ViPER with the attributes requested as name-value pairs (step 3):

```
resourceID = {23, 42}
name = !"IXUS"
model = ?Model
price = ?Price
resource = ?Resource
```

(4.3)

Every string that occurs in the input is marked with an exclamation mark (!), to indicate ViPER that it can be used for query generation to fill out search forms. On the other side, variables marked with a question mark (?) indicate the annotated data items we are interested in. ViPER internally resolves the resource ids to valid Web URLs and extracts the desired data records. With a new extension of ViPER we are also able to annotate these data records according to our background ontology (step 4). The resulting bindings are returned to the Communication Interface with the resource ids resolved to URLs (step 5):

```
[resourceID = 23
 resource = "http://www.amazon.de",
 name = "IXUS",
 model = "Digital Ixus 60",
 price = 269],
[resourceID = 42
 resource = "http://www.mediaonline.de",
 name = "IXUS",
 model = "Digital Ixus 700",
 price = 295]
```

(4.4)

In listing (4.4) each resource only contributes one result, but generally multiple results per resource are usual. These results are converted to F-Logic facts and inserted into our ontology.

```
...
ixus[name→"IXUS"; resourceID→{23, 42}].
ixus_1 : ixus[model→"Digital Ixus 60"; price→269;
           resource→"http://www.amazon.de"].
ixus_2 : ixus[model→"Digital Ixus 700"; price→295;
           resource→"http://www.mediaonline.de"].
```

(4.5)

Listing (4.5) shows the final state of our ontology instantiated with the extracted data items. The new instances have been inserted into the fact base as instances of the respective class, i.e. `ixus`, inheriting the `name` attribute from it with the value set to "IXUS". The instances could either be stored for later time series

analysis or the ontology will be reset to its initial state after having finished the query.

Now we are able to select the attributes of interest, taking into account the information from the "Result properties" form field in figure 1, resulting in the following F-Logic query (step 6):

```
_Obj:ixus[price→Price; model→Model;
          name→"IXUS"; resource→Resources], Price < 270.
```

 (4.6)

The Florid results are shown in listing (4.7) and are finally transformed into a tabular HTML representation, where we additionally list the resources that provided the data items (step 7).

```
Resources/http://www.amazon.de,
Price/269,
Model/"Digital Ixus 60"
```

 (4.7)

5 Conclusion

We presented the OntoGather system, an ontology-based dynamic Web resource querying engine, that is geared towards the requirements of a dynamic world. Because of our resource preselection mechanism we are able to process a user query from a pool of different resources that we decide on at runtime. This is made possible by our fully-automatic Web data extraction system ViPER. Our main contributions are twofold: first the selection of the query-relevant resources and second the reasoning on fresh data items extracted and annotated by ViPER, which both happens at query time.

6 Outlook

Our future goals include time series analysis, which is explicitly supported by our object-centered approach, by aggregating the results of several user queries. This could be realized by introducing a method `time_stamp` to indicate the freshness of the information. Additionally including ECA rules might be an interesting topic while monitoring specific dynamic resources over a given time frame for an invariant query.

References

1. Frohn, J., Himmeröder, R., Kandzia, P.T., Lausen, G., Schleppehorst, C.: FLORID: A Prototype for F-Logic. In: ICDE'97, IEEE Computer Society (1997) 583
2. Kifer, M., Lausen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM* **42** (1995) 741–843
3. Baumgartner, R., Flesca, S., Gottlob, G.: Visual Web Information Extraction with Lixto. In: VLDB. (2001) 119–128
4. Simon, K., Lausen, G.: ViPER: Augmenting Automatic Information Extraction with Visual Perceptions. In: ACM CIKM'05, Bremen, GERMANY, ACM Press (2005) 381–388