Catalog Search Engine: Semantics applied to products search

Jacques-Albert De Blasio, Takahiro Kawamura, and Tetsuo Hasegawa

Research and Development Center, Toshiba Corp.

Abstract. The Semantic Web introduces the need for semantic search engines. In this paper, we explain our vision of a catalog search engine for semantically defined products. With our prototype, we address the problem of products' information retrieval over the Internet and their semantic enrichment through the mixed usage of thesauruses and ontologies. We show how we automatically build a repository of instances of ontology classes, and how we dynamically prioritize the search variables of our engine. We then introduce our prototype which, through the use of all those concepts, improves the user experience.

1 Introduction

The Semantic Web introduces the need for semantic search engines. Although semantic search is already available in a variety of forms such as SHOE[1] or Ask Jeeves[2], semantic search for products sold on the Internet is rarely available. With the system we developed, we strived to fill this gap.

Products catalogs available on the Internet all have limitations of several types. They either provide a wide range of products but have a poor search engine in terms of precision, or offer a limited range of products with a powerful but too specialized (in terms of search variables) search engine. Whichever the catalog, the user can easily get frustrated by their poor ability to supply him/her precise results and an extensive selection of products at the same time.

One of the challenges of the semantic web is to transform the already available information into more meaningful, more usable data. A lot of the available literature tackles this problem and agrees on a fundamental problem: most of the difficulties come from the ambiguity of the human language, and from the fact that nearly all this information has been created by humans for human consumption. Products information, on the other hand, has the advantage of being, in most cases, based on an agreed vocabulary. However, the problem with products sold on the Internet is that the quality of their descriptions (in terms of availability), as well as the presentation of those descriptions (in structured tables, simple paragraphs, etc) varies greatly from a web site to another. Nonetheless, considering that the Internet is the biggest products database available, it becomes obvious that it should be the source of any search engine aspiring to be as complete and accurate as possible.

Our vision of a catalog search engine includes three distinct goals. The catalog must contain as much products as possible, its search engine must be the most

accurate, and the user must be given enough tools to let him/her search efficiently through the catalog. In this paper, we show that we answered to those three needs with the following strategies. The wideness of the catalog is insured by the gathering of existing products' information all over the Internet through the usage of dedicated parsers. The accuracy of the search engine is reached by a combination of semantic enrichment of the previously fetched information, and the automatic conversion into logic facts of every characteristic of the products. Eventually, the user's search efficiency is enhanced by the usage of algorithms dynamically computing the usefulness of each products' characteristic. Moreover, the usage of a thesaurus during the query phase allows the user not to worry about the exactitude of the content of his/her query.

In the following sections, we first introduce the architecture of our system. Then, we focus on its usage and explain the details of its features. We continue with a short demonstration of our prototype and follow with a discussion about decisions we took during the design phase. Eventually, we take a brief look at the existing work tackling the problem of catalog search engines and conclude.

2 Architecture of the Catalog Search Engine

2.1 Overall Architecture

The architecture of the catalog search engine is shown in Fig. 1. The system we suggest is a complete solution, from the server fetching data from the Internet, to the client that will let the end-user carry out his/her requests. The server is made of 4 main components; the web page fetcher, the parser, the facts creator and the profiles creator. The client is separated into 2 main components; a GUI with which the end-user will communicate, and a proxy which will be in charge of the client-Matchmaker communication. In between the server and the client lies the Matchmaker server which provides the search capabilities.

The main idea of this system consists in fetching information about products sold on the Internet, and publishing this information on the Matchmaker server. This information will be enriched with semantics using ontologies. On the other side, the end-user will be able to express requests to the Matchmaker. The latter will browse through all the available advertisements, try to find those which are the most closely related to the request and eventually return them to the user. We will describe the usage in the next section.

At the heart of our prototype lies the Semantic Service Matchmaker, a service search engine based on the LARKS[3] algorithm. It adopts the filtering approach which uses sophisticated information retrieval mechanisms and ontology-based subsumption mechanisms to match requests against advertisements. This engine has already proven to be efficient in regard to web services matchmaking[4][5].

Ideally, when the requester looks for a product, the Matchmaker will retrieve a product that matches exactly the expected one. In practice, if the exact product is not available, the Matchmaker will retrieve one which capabilities are similar to those expected by the requester. Ultimately, the matching process is the result of the interaction of the products available, and the requirements of the requester.

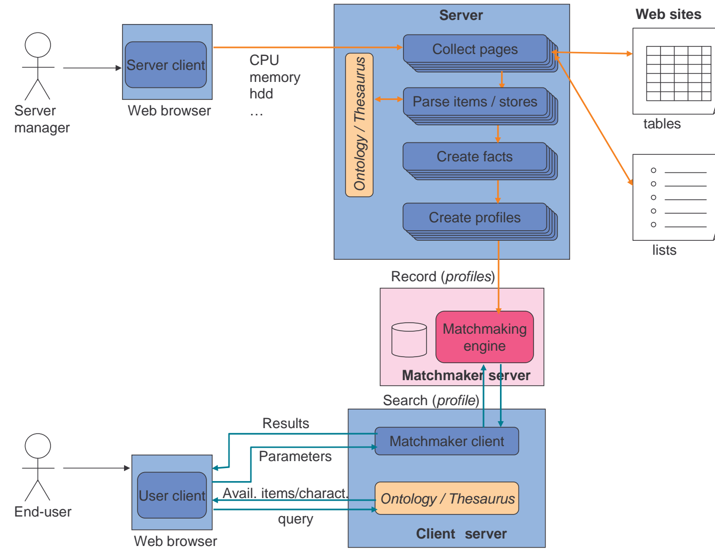

Fig. 1. Architecture and flow of the system. Note that the "ontology / thesaurus" denotes the same instance on the server and client sides

Although the Matchmaker originally provides a set of 5 filters, our prototype uses only two of them. The *type* filter applies a set of subtype inferencing rules mainly based on structural algorithm to determine whether an ontology class is a subsumption of another. The *constraint* filter has the responsibility to verify whether the subsumption relationship for each of the constraints are logically valid. The Matchmaker computes the logical implication among constraints by using polynomial subsumption checking for Horn clauses. More details about the Matchmaker's filters are provided in [4].

2.2 Usage scenario

Server side

1. The server is initialized with a file which contains information concerning the web pages to be fetched and parsed. This file connects each type of products with a list of web pages (e.g `http://somewhere/hddidetosell.html` ⇒ products type "HDD IDE").

2. Next, web pages are fetched from selected web sites. Once done, a parser detects relevant information from those web pages. If a new product is detected, the server automatically creates a new instance of the ontology class which describes the type of the product. If a new characteristic is detected, the server updates the list of properties associated with each class of product.

3. Then, the server automatically creates a file containing a list of facts written in RDF-RuleML[8][9]. Each fact corresponds to a characteristic of a product (see section 2.4).

4. Eventually, the server creates an "advertisement" profile for each product. A profile is the semantic description of a product (see section 2.5). Once all the profiles for all the products have been created, they are registered in the Matchmaker server.

Client side

1. First the user inputs a query. This query is parsed and its content is compared with the thesaurus' words, as well as with the name of the instances of the products' types (created at step 2 of the server part).

2. The answer to the query is either a list of types of products having the best matching terms to the query, or a list of instances, or both. If the answer is a list of instances, the user can click on one of them in order to display the details of the chosen product. If the answer is a list of types of products, the user can click on one of them in order to display a list of characteristics of the chosen type. If the user wants to carry out a fine search, he/she must input some values for the characteristics with which he/she wants the result to be relevant. When the user eventually clicks on the "finer search" button, a RDF-RuleML file containing those characteristics translated into facts is automatically created.

3. The system automatically creates a "request" profile. This profile is then submitted to the Matchmaker, which tries to match this "request" to the "advertisements" contained in its database. If one or more matching profiles are found, they are sent back to the user, who sees them as individual products. He/she can then click on one of the available link to different shops selling the product.

2.3 Usage of thesauruses and ontologies

Thesauruses are needed for any search engine of which search mechanism is not exclusively based on the query's keywords. For the sake of simplicity, we built our own thesaurus for our prototype. While the goal is, of course, to use a rather complete thesaurus such as WordNet[6], we wanted our thesaurus to be multilingual, which WordNet is not. In our thesaurus, n terms can be synonyms of n other terms, and each term is translated into m languages. As the user can choose in which language he/she would like to interact with our prototype, it will set the language of the thesaurus. In the future we want to improve this by letting the user type the request in any language and let the search engine browse through the entire thesaurus, without any preference for the language.

Our ontologies are written in OWL[7]. An OWL class corresponds to a product type which characteristics are described using the OWL properties. Each property's *range* is either an object (e.g. the type of interface of a hard disk) or a value (e.g. the capacity of a hard disk, in gigabytes). The Fig. 2 shows our internal representation of an ontology. The "datatype" attribute is needed in order to make sure that the value of a product's characteristic taken from a web site (during the parsing) has the same type as the one expected.

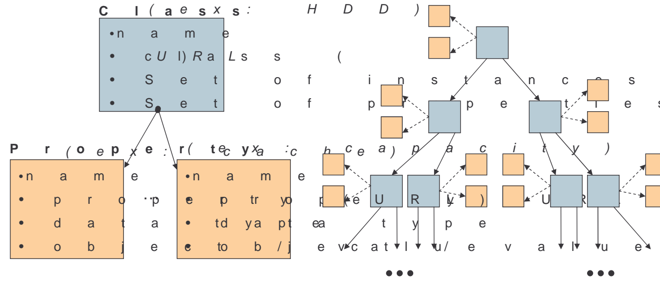

Fig. 2. On the left: internal representation of the ontology. On the right: architecture of our ontologies (note that the relationships are not necessarily binaries).

In our system, we have a general ontology composed of classes corresponding to various types of products (e.g. “hdd ide”, “scanner”, etc). We assume that this ontology has been created prior to the launch of the system. When the server of the system browses and parses the information found on various web sites, it automatically fills in each instances slot for each class. For example, all the products found as being of type “hdd ide” will be automatically added to the instances slot of the product type (the class) “hdd ide”. The instances growth is non-monotonic. It makes sense in that products may disappear if the catalog of the sellers is updated. In our prototype, we track each product’s availability, so that if all the web sites of which we get information do not offer a product anymore, its instance is automatically deleted.

A problem we encountered was related to the names of the products. A product has, in general, the same name wherever it is sold. Even so, some web sites on the Internet mix the name and the characteristics of the products in the same string. The instances we record being based on the name, we had to insure that, even though the name may be different for a same product, new instances might not be created. To solve this problem, we first check the manufacturers of the products. If they match, we use a parser which is able to correct suspicious products’ names through string comparisons. Also, as some products may have different names in different countries, we use a thesaurus of products’names to make sure that the products are the same.

2.4 Characteristics of the products

A product is distinguished by its characteristics. A hard disk has a certain seek time, capacity, interface, etc. Translating products’ characteristics into facts is quite natural, as each characteristic can be thought as a *truth* about the product it describes. The facts our system creates are all of the form $P(x, y)$ where P is a predicate, and x and y two terms. The facts written by the server use two kinds of predicates, “equal” when the term y is a numerical value, and “is” when the term y is a string of characters. The facts written by the client use the predicates “is less than or equal to” and “is more than or equal to” when y is a numerical value, “is” when y is a string of characters. Each property of

the products' classes of our main ontology are converted into OWL classes and used as the term x of the facts (for a discussion about this conversion, refer to section 4). The facts concerning the manufacturer, the seller and the price of a product are considered the minimum information required for a product to be taken into account.

In the table 1, we show an example of a product's characteristics converted into facts on the server and the client side. On the server side, the characteristics of a hard disk fetched from the Internet are converted into facts. On the client side, characteristics which values have been input by the user are also converted into facts. "MANUFACTURER", "COST", "SELLER" and "CAPACITY" were OWL properties converted into classes. The predicates are classes of an ontology describing predicates. We use our own thesaurus to make the connections between the words used in the description of a characteristic (e.g. *"sold at"*) and the corresponding ontology class (e.g. "COST"). See the code at section 2.5 for an example of a fact.

Server side	
Char. fetched from web sites	Facts
sold at **45,000** Yen	*equal* (COST, 45,000)
manufactured by **Toshiba**	*is*(MANUFACTURER, Toshiba)
sold by **anotherMart**	*is* (SELLER, anotherMart)
a capacity of **80** Gb	*equal* (CAPACITY, 80)
Client side	
Char. with values	Facts
price $\leqslant$ **50,000**	*is less than or equal to* (PRICE, 50,000)
capacity $\geqslant$ **60**	*is more than or equal to* (CAPACITY, 60)

Table 1. Characteristics translated into facts - Server side

Once the client has transmitted the request profile (which contains links to the facts created by the client) to the Matchmaker, the latter will use its inference engine (called the *constraints* filter) to match the facts of the advertisements to the facts of the request.

2.5 Profiles

A profile is an OWL file containing a semantic description of a product, as well as a list of links to each fact present in the facts files related to this same product. For each shop selling the product, a profile is created (i.e. a product being sold by 10 shops will have 10 different profiles). The information stored in those profiles are the ontology class of the product's type, the name of the product (only if the profile is an advertisement), the list of facts and the URL to the shop selling the product. Once advertisements profiles have been registered to the Matchmaker, if a request profile is submitted the Matchmaker applies a matching using its *type* filter on the ontology class of the product's type and its *constraint* filter on all the facts. The following code shows an example of a profile and a fact.

```
<product:description rdf:id="Kakaku_CPU_Athlon_64_2800_Socket754_5">
    <product:name>ATHLON 64 2800 Socket754_5</productName>
    <product:restrictedTo rdf:resource="http://somewhere/onto.owl#cpu" />
    <product:constraint rdf:resource="http://somewhere/facts.rdf#clockspeed" />
    <product:constraint rdf:resource="http://somewhere/facts.rdf#cost" />
    <product:constraint rdf:resource="http://somewhere/facts.rdf#manufacturer" />
    ...
    <product:shopURL> http://www.aShopURL.com/</product:shopURL>
</product:description>
```

```
<ruleml:Fact ruleml:label="cost">
  <ruleml:head>
    <ruleml:Atom ruleml:rel="http://somewhere/predicates.owl#numericallyEqual">
      <ruleml:args>
        <rdf:Seq>
          <rdf:li>
            <ruleml:Var ruleml:name="http://somewhere/store.owl#COST" />
          </rdf:li>
          <rdf:li>
            <ruleml:Ind ruleml:name="20990" />
          </rdf:li>
        </rdf:Seq>
      </ruleml:args>
    </ruleml:Atom>
  </ruleml:head>
</ruleml:Fact>
```

2.6 Dynamic update and prioritization of the characteristics

When the user searches for a given type of product, its related characteristics are displayed. If the user wants to carry out a fine search, he/she can insert some values to the characteristics he/she wants to be respected. The list of available characteristics is updated on the server side, when fetching and parsing information from various web sites. However, the priority in which those characteristics are shown to the user is dependant of each characteristic's associated weight. The weights are updated as follows.

- the more a characteristic is available for a given type of product, the greater its weight will be,
- the more a characteristic has possible values, the greater its weight will be (e.g. the size of a screen can be 15", 17",19", 21", etc),
- the more a characteristic is chosen by the user, the greater its weight will be.

Other conditions come also into play to determine the position of a characteristic. As products' types are ontologically defined, we rely on the parent-child relationships to tell whether a product's characteristic should be shown before another. For instance, as "computer" is the parent class of "notebook computer", if the user searches for "notebook computers" its characteristics should be displayed prior to those of "computer".

3 The Prototype

We introduce here the prototype of the client application. Data from the Internet has already been fetched from Kakaku[10], a Japanese catalog web site, and parsed on the server side.

In Fig. 3(a), the user entered “hard disk” as a query. With the thesaurus, the client found out that “hard disk” is the equivalent to “hdd”. Moreover, using the ontology, the client proposes not only “hdd” but also “hdd ide” and “hdd scsi”, two subclasses of “hdd”. In Fig. 3(b), the user selected “hdd ide”. The client now proposes a list of characteristics of the “hdd ide”. The three first are always present for each product. The next ones are the characteristics available for “hdd ide”, as well as “hdd”, as well as any other parent class of the “hdd”, back up to the root of the ontology. The user decided to input values for two characteristics, the cost and the capacity. Once done, he/she gets the result shown in Fig. 3(c). The values corresponding to the chosen characteristics in the previous step are shown in bold. A link is provided to shops selling the products.

4 Discussion

Our intention was not to produce a very efficient catalog search engine in terms of speed, but rather in terms of relevance of the results. In this regard, we reached the three goals cited in the introduction of this paper. As our system gathers data from various web sites, we get more details about the products than if we simply relied on specific vendors and thus insure the wideness of the catalog. Accuracy of the search engine is provided by the combined usage of thesauruses and ontologies, allowing the system to return very precise results even if the query of the user is relatively vague. Eventually, search efficiency is attained by handling the feedback of the user regarding the characteristics of the products. As a consequence of all this, we observed that a user needs about half the number of clicks than usually needed when accessing the same information about products on other web sites such as Kakaku.

However, as our system is still at the stage of a prototype, it is not without flaws. In fact, the parser approach to the problem of information fetching on various web sites can prove to be quite weak in the long term. It requires much more advanced techniques to be able to fetch facts or rules from web pages such as Amazon or Yahoo Shopping as those web sites do not always display information about products in a very formal way.

The reader may wonder why we chose to create facts using RDF-RuleML to describe the characteristics of each products, instead of directly using the properties of each products’ classes of our ontology. The reason is that we intend to create a much more powerful search engine, which does more than giving the possibility to enter a value for each characteristic. The goal is to use a better Natural Language Processing tool during the parsing phase on the server side, so that the system becomes able to create rules such as “*if the credit card is Visa or American Express, the customer can have a 5% discount*”. This kind of rule can not be expressed with OWL’s classes and properties.

Alternatively, we thought about expressing the facts and rules in SWRL[11] instead of RDF-RuleML. The advantage is that SWRL allows the use of properties which have been created in the ontology, and thus avoids redundancy. However, the terms of an atom in SWRL must be either variables, OWL indi-

viduals or OWL data values. Unfortunately, as individuals lack any subsumption relationship, the constraint filter of the Matchmaker would not work efficiently.

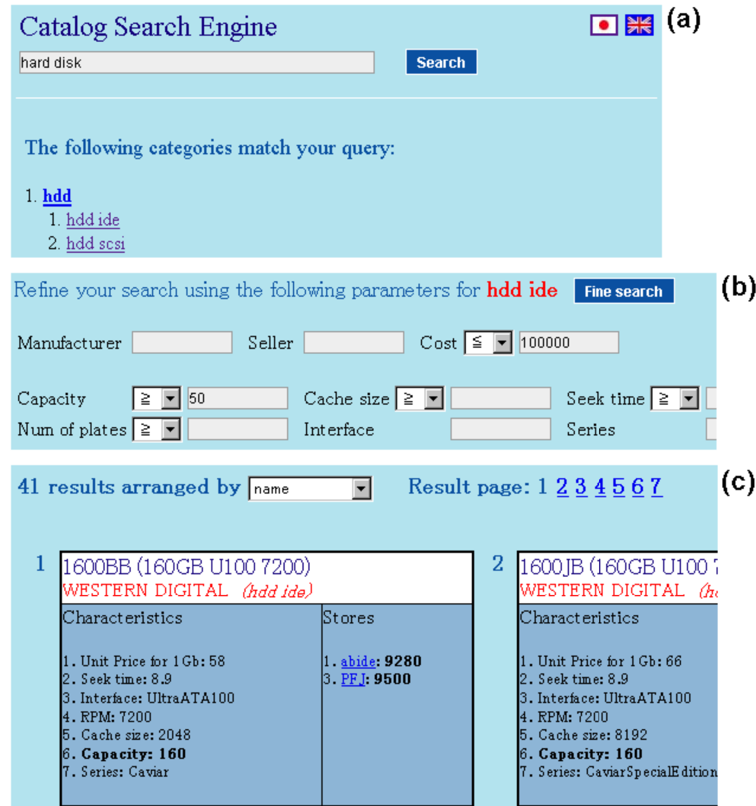

Fig. 3. Prototype screenshots. Three steps to search for products (cropped images)

5 Related Work

Froogle, a twin of Google in a shopping search engine point of view, offers a wide catalog and blatant speed, but allows search refinement only through price range. Kakaku gives the possibility to search using the products' characteristics, but the number of the latter is static. Both search engines get the products information directly from the vendors. Although it insures accuracy, this method limits greatly the number of sources of information. Amazon is too restrictive in terms of products, as they propose only those which they sell. To our knowledge, none of the web sites cited above make use of semantics.

The IWebS project[12][13] aims at creating an intelligent yellow pages service with semantically annotated services. Although they share some similarities with

our approach, they introduce the needs for manual annotations which would be intolerable for a database of thousands of different products.

Active Catalog[14] focuses on how retrieved information can be used to engineer parts and physical objects. Its database is entirely built beforehand, that is, there is no dynamic data acquisition. The parts' characteristics are also all predetermined. Eventually, the content as well as the usage makes it usable exclusively to engineers.

6 Conclusion

Based on the Matchmaker, we developed a prototype of a catalog search engine which enables users to have more accurate results in regard to their queries. Search parameters are dynamically updated through the analysis of fetched information and the feedback from the users. We showed that the approach of fetching available products data from the Internet, adding semantic to it through the use of ontologies, and efficiently searching through it is feasible. Using rules, our system will be able to give more expressive power to users' queries.

References

1. Simple HTML Ontology Extensions, http://www.cs.umd.edu/projects/plus/SHOE/
2. Ask Jeeves, http://www.ask.com/.
3. K. Sycara, S. Widoff, M. Klusch, J. Lu, "LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace", In Autonomous Agents and Multi-Agent Systems, Vol.5, pp.173-203, 2002.
4. M. Paolucci, T. Kawamura, T. R. Payne, K. Sycara, "Semantic Matching of Web Services Capabilities", Proceedings of First International Semantic Web Conference (ISWC 2002), IEEE, pp. 333-347, 2002.
5. T. Kawamura, J. D. Blasio, T. Hasegawa, M. Paolucci, K. Sycara, "Public Deployment of Semantic Service Matchmaker with UDDI Business Registry", Proceedings of 3rd International Semantic Web Conference (ISWC 2004), 2004. to appear.
6. WordNet, http://www.cogsci.princeton.edu/ wn/
7. Web Ontology Language, http://www.w3.org/TR/owl-ref/.
8. Resource Description Framework, http://www.w3.org/RDF/.
9. RuleML, http://www.ruleml.org/.
10. Kakaku, http://www.kakaku.com.
11. Semantic Web Rule Language, http://www.w3.org/Submission/SWRL/.
12. M. Laukkanen, K. Viljanen, M. Apiola, P. Lindgren, and E. Hyvonen. "Towards Ontology-Based Yellow Page Services". In Proceedings of the WWW 2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web, New York, USA, May 18th 2004.
13. E. Hyvonen, K. Viljanen, A. Hatinen. "Yellow pages on the semantic Web". Towards the Semantic Web and Web services. In Proceedings of XML Conference, Finland 2002.
14. S.R. Ling, J. Kim, P. Will, and P. Luo, "Active Catalog: Searching and Using Catalog Information in Internet-Based Design," Proceedings of DETC '97 - 1997 ASME Design Engineering Technical Conferences, Sacramento, California, September 14-17, 1997.