# Generating Advanced Query Interfaces

Dongwon Lee Divesh Srivastava Dimitra Vista

**Abstract**

With the increasing popularity of the World Wide Web, the number of information sources providing access to various types of data has increased considerably. While simple data retrieval queries are typically very easy to specify in such systems, formulating advanced queries is much harder. Most systems require the user to be aware of their specific, often proprietary, interfaces and query languages. We believe that being able to query these systems in a uniform and consistent way could greatly improve the user's experience of interacting with these systems.

In this paper, we present an interface generator for creating advanced query interfaces. Our tool allows the easy construction of advanced interfaces, with a consistent look and feel, to different information sources, removing the burden of users having to know the interface and query language specifics of individual information sources. We demonstrate the power of the interface generator by using the tool to generate advanced interfaces for various Web search engines (AltaVista, Excite, HotBot, and Infoseek), directories (Four11, Bigfoot, LDAP), and specialized search tools (IBM's patent server, DejaNews).

## 1 Introduction

Many information sources available on the World Wide Web support query interfaces to allow users to select subsets of the data in the information sources that are of interest to them. We focus our interest on the subset of the Web that is accessible via such query interfaces. The examples of such systems are numerous: web search engines, white and yellow pages, special purpose databases with articles, patents, movies, stock quotes, software, flights, cars, and many more. Many of these systems allow access to their data from multiple query interfaces typically including a) an interface based on a simple vector-space model [Sal89] for limited keyword searching, and b) a more advanced interface for specifying complex selection criteria on the data to be retrieved. While simple queries are typically very easy to specify, writing complex queries is much harder. Most systems require the user to be aware of their specific, often proprietary, interfaces and query language syntaxes. Having interfaces that hide the idiosyncrasies of their query languages can significantly improve the user's experience in interacting with the various information sources.

In this paper we describe how we generate easy to use advanced query interfaces for various information sources by using an interface generator tool. The tool accepts as input the specification of the interface to generate and produces as output the implementation of the specified interface. In general, constructing intuitive and easy to use visual interfaces for general query languages is a difficult problem [KZ95]. Fortunately, most information systems on the Web do not support general query languages: they only allow data to be *selected*. We observed that there is a particular way to look at the data, and consequently at selection queries, which allows the easy construction of interfaces with a uniform look and feel. In this paper we exploit this abstraction towards this goal.

There are two aspects to each visual interface generated by our tool. The first has to do with the layout of the various visual components in the interface and the way they interact in response to user actions. The second has to do with the way the visual interface supports the language interactions between itself and the back-end system. This interaction includes the mapping from the queries constructed in the visual interface to queries supported by the underlying system, and the mapping from individual query elements of the underlying language to visual elements of the interface.

Our goal in devising the generator tool was not to study which visual layout would be best suited for the kind of applications we are targeting. For this, reason all interfaces generated by our tool share the same layout of visual components. Our goal was to provide a configurable tool with respect to language interactions. We have identified many customizable options that a generator tool can support with respect to language interactions and our current implementation supports most of them.

The rest of the paper is organized as follows. In Section 2 we elaborate on the basic idea behind the advanced interface generator tool. In Section 3 we present the layout of the visual components which is common among all of the interfaces produced by the generator. In Section 4 we discuss aspects of configurability that the generator is amenable to and in Section 5 we present the format of the input configuration file for interface specification. In Section 6 we discuss how we have applied the generator tool to devise advanced interfaces for web search engines, directories and other specialized information sources. We conclude in Section 7.

## 2 Basic Idea

To demonstrate the basic idea behind the tool described in this paper, we use examples from the application of web searching. Suppose we were looking for all web documents containing the keyword `WWW7` in their title. We can think of each web page as a structured document having a title, a body, a domain, one or more headers and a number of other such *attributes*. The query simply asks for all documents whose `title` attribute `contains` the value `WWW7`. As another example, if we were looking for web documents not containing the phrase `call for papers` in their body, we could ask that the `body` attribute `not contain` the value `call for papers`.

Many selection criteria can thus be expressed as a combination of an attribute of the document to be retrieved, an attribute operator to be evaluated on it and a given value. We call such selections *atomic* queries. Atomic queries are logically combined to form *complex* queries. As an example, suppose we were interested in all documents containing the word `WWW7` in their title but not containing the phrase `call for papers` in their body. We can express this complex query as the logical AND of the two atomic ones discussed above.

Web search engines differ in the way they express complex queries. Our example is expressed as follows in the Alta Vista and HotBot search engines:

| | |
|---|---|
| `Alta Vista` | `title:WWW7 AND NOT ‘‘call for papers’’` |
| `HotBot` | `title:+WWW7 −‘‘call for papers’’` |

If we use the document model described above we can formulate the same query with a description such as:

```
title contains word WWW7
so long as
body does not contain phrase call for papers
```

This expression does not have to be specified as such by a user – it could simply be an English language description of the output of a graphical interface. However, if we wanted to utilize this description in both search engines, we would ultimately have to translate it into the expressions understood by these systems. The observation is that we can construct a graphical interface that allows the specification of expressions like this and translates them into expressions of a search engine's query language, ultimately providing a uniform interface for querying both systems.

Once we recognize that similar interfaces can be built for a variety of sources, a question of interest becomes whether there are enough commonalities between them so that we can abstract away from the details of each and devise a core system to be shared by all. The attribute/value model presented here gives us a modeling tool to achieve such an abstraction. In addition to the conceptual model, the interfaces can share the same visual layout of components on the screen, the mechanisms by which the various components of the interface interact with each other in response to user actions, as well as internal representations and manipulations of the queries they construct.

In the next section we describe the visual layout shared by all interfaces generated by our tool and explain our reasoning for choosing it.

# 3 Visual Component Layout

Figure 1 shows a typical interface generated by our tool. The screen dump is from the HotBot search engine interface showing a query requesting all web documents on the Intel Pentium Chip outside of the intel.com domain. The interface contains the following visual components:

- The upper part consists of a pop-up menu of attribute names, a pop-up menu of attribute operators and a place holder for entering the value of an atomic query. In other words, the upper part is dedicated to atomic querying. The figure only shows one item of the pop-up menu for the attribute names (`HTML Page`) and attribute operators (`contains word`).
- The horizontal pallete of buttons underneath corresponds to the logical connectives one wishes to support for complex querying. Three are shown in the figure: `so long as`, `also`, and `except`.
- The vertical pallete of buttons to the right corresponds to functionality shared amongst all generated interfaces. They permit editing of the generated expressions.
- The central textual component displays an English language description of the constructed query. Individual constructs can be selected for editing, removing and so on.
- The (optional) lower textual component displays the generated query in the syntax of the underlying system.

In devising this particular layout for the interfaces generated by our tool we made the following decisions. First of all, we wanted the interfaces to report to the user the meaning of the query constructed so far in an intuitive way. Many query languages currently supported by information sources are understood easily by technically oriented people. But as the Web is being used more and more by people without much technical background, native query languages become inadequate. An English language description of the query seemed an appropriate alternative representation. Parenthetically, this was also the reason for choosing the words `so long as` and `also` for the logical

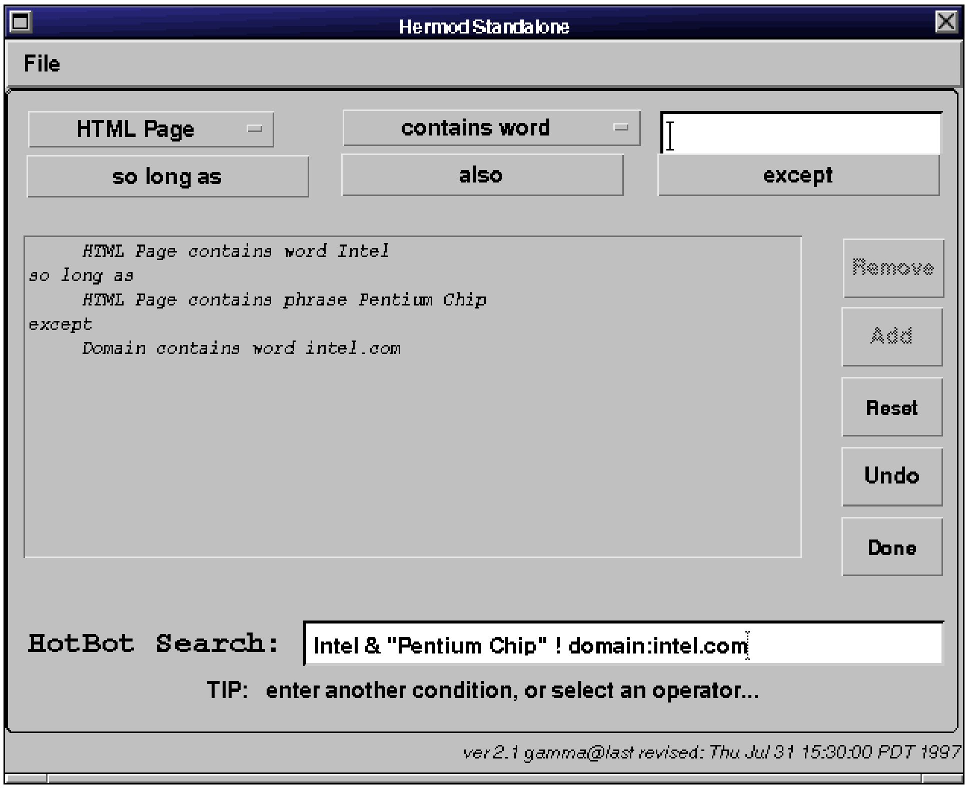


Figure 1: A typical interface generated by the generator: HotBot query requesting documents on the Intel Pentium Chip.

AND and OR, respectively: in English, the word 'and' sometimes denotes a logical disjunction, while the word 'or' typically denotes an exclusive disjunction!

We also wanted to separate the concept of visualizing the description of the query (the central component) from its editing (the upper component) and general manipulation (performed mostly by functions of the rightmost component) which explains the three main panels of the layout.

Finally, we wanted the uppermost panel to be used in place for the editing of individual atomic queries. This is in contrast with many interfaces we find on the Web, where a user has to press buttons for more or less options. There were two reasons why we did not follow the common pattern. Besides being somewhat cumbersome, an extendable list of editable triplets for the attribute, the operator and the value could only be used for conjunctive queries while we clearly wanted a solution that coult accommodate disjunctive and other complex queries as well.

Although our generator tool imposes this particular layout of visual components, it can be customized in many different ways. In the next section we talk about what we are trying to customize.

# 4 Tool Configuration

The data abstraction explored in this paper views data being queried as objects described by a set of attribute/value pairs. An attribute of the object to be retrieved, an attribute operator to be evaluated on it and a given value form an *atomic* query. Atomic queries are combined using complex operators to form *complex* queries. We configure the generator tool with respect to the generation and tranlation of such queries.

### Query Construction

Customizing the way queries can be formulated in the visual interface includes specifying a) how the attributes, attribute operators and complex operators we choose to support for a given application are visually represented on the screen, and b) what types of queries we can formulate in the interface by using them. In terms of restricting the types of queries, in our current implementation we can specify which attribute operators are applicable to each attribute. We can also specify the associativity and precedence of complex operators. For example, a left to right associativity of the operators AND and OR, with AND having a higher precedence, allows the easy specification of queries in disjunctive normal form. At the current time, no explicit parenthesization is supported: specifications of operator precedence and associativity are the only way to control what parenthesizations are favoured in the interface. These specifications reflect on the way the English language description of the query is indented.

Other aspects of query construction configurability may include the specification of complex operators with arbitrary arity. Our current implementation only supports binary operators.

### Query Translation

Customizing the translation of queries refers to the specification of the mapping from queries in the interface to queries in the underlying system. In the current implementation we allow for the following flexibility in supporting query translation: we can specify how attributes, attribute operators, atomic queries and complex queries are translated. The translation function of an attribute is part of the specification for the attribute. The translation function of an atomic query is part of the specification of the attribute operator involved in the query. The translation function of a complex query is part of the specification of the complex operator involved in the query. We implicitly assume that specified values are trivially translated into themselves.

In general, we might want to support more flexible translation functions. For example, we might want the translation of an atomic query to depend not only on the attribute operator involved by also on the attribute and/or the specified value. Similarly, we might want the translation of complex queries to depend not only on the complex operator involved but also on one or more of its arguments. Finally, we might want to utilize non-trivial translation functions for the values themselves.

### Data Semantics

All options discussed so far are related to the syntax of queries. Additional configurable options are available to help incorporate data semantics of the application into the generated interface. To accomodate data semantics we support attribute types and attribute default values. By declaring that an attribute is of a particular type, we can generate advanced interfaces capable of type checking that values entered satisfy the type constraint of the corresponding attribute. We also support default attribute values to populate the place holder for the value when the attribute is selected from the attribute menu. Such default values can be used as hints to help users enter a proper value. For example, when an attribute of type `date` is selected, the place holder can have a value of the form `dd/mm/yy` suggesting this particular format for value specification. To accomodate the case when attributes can take values only from a given set of possible values, we allow the specification of such sets. In this case, the interface dynamically replaces the value holder with a menu of available options or appropriate radio buttons.

Additionally, we can specify more general types of integrity constraints on the values of a particular type. For example, we can specify that the value of an attribute must be positive, less than a given threshold and so on. Our tool does not support such constraints.

In the next section we describe in detail the format of the input configuration file as currently supported by our generator tool.

# 5 Format of the Configuration File

First of all, in the configuration file we specify the (non-empty) set of attributes to appear in the attribute menu. For each attribute, we specify how the attribute is visually displayed, and how it is translated into the query language of the underlying system. For example, the following specification says that there is an attribute that appears as `'Title'` and is translated to `'title:'`.

*Attribute ::* `'Title'` *is* `'title:'`

Sometimes, we want to specify a default value for an attribute: when we choose the attribute from the menu, the default value appears in the place holder for the value. For example, the following specification says that the attribute `'URL'` has default value `'http://'`.

*Attribute ::* `'URL'` *is* `'url:'` *withvalue* `'http://'`

Our generator tool supports a number of predefined types, including `string`, `int`, `tel` (telephone number), `date`, `booleanof` and `choiceof`. The types `booleanof` and `choiceof` are used to restrict the values that an attribute can take. In particular, `booleanof` presents the user with two values as radio buttons while `choiceof` presents a preselected set of values as a choice menu. Specifying that an attribute is of a given type helps the generator produce interfaces with embedded type checking capabilities. The type `string` is the default type. Here are some examples.

| | | |
|---|---|---|
| *Attribute ::* | `'Telephone Number'` | *is* `'tel'` *withtype* `'tel'` |
| *Attribute ::* | `'On vacation'` | *is* `'onvacation'` |
| | | *withtype* `booleanof('true', 'false')` *withvalue* `'false'` |

The first example declares the telephone number attribute to be of type `'tel'`. The second example restricts the values for the `'On vacation'` attribute to only `'true'` and `'false'` to be displayed using two radio buttons. The button for `'false'` is initially checked.

In addition to attributes, we specify the (non-empty) set of attribute operators for the attribute operator menu. For each attribute operator, we specify how it is visually displayed and how it is translated into the query language of the underlying system. For example, the specification

*Attribute Operator ::* `'must contain phrase'` *is* `'+$attr''$val'''`

says that there exists the `'must contain phrase'` attribute operator which is translated by using the translation of the attribute to which it is applied, prefixed by the symbol '+' and suffixed by the quoted translation of the value specified. The variables `$attr` and `$val` are bound dynamically whenever a user selects an attribute from the attribute menu or specifies a value in the value holder.

All attribute operators are applicable to all attributes by default. Sometimes we might want to restrict this convention by a different applicability of attribute operators. For example, the following specification says that only operators allowed on the `'On vacation'` attribute are `'is'` and `'exists'`. Upon selecting the attribute all other choices are greyed out.

*Attribute ::* `'On vacation'` *is* `'onvacation'` *withtype* `booleanof('true', 'false')`
*withvalue* `'false''` *limitedto(*`'is', exists'`*)*

Finally, in the configuration file we specify the set of logical operators for the horizontal operator palette. For example, the following specification says that there is a complex operator that appears as `'so long as'` and is translated using the translations of its two operands with a '&' in between. The variables `$lhs` and `$rhs` are bound dynamically to the two operands of the operator.

*Complex Operator ::* `'so long as'` *is* `'$lhs` *&* `$rhs'`

Having described the syntax of the configuration file, we can now describe how easily we have used such specifications to generate advanced interfaces for various information sources.

# 6 Applications

## 6.1 Web Searching

We have used our tool to devise advanced interfaces for various search engines including AltaVista [Alt], Excite [Exc], HotBot [Hot], and Infoseek [Inf]. We only discuss the HotBot search engine here. Focusing on only one system is enough to understand the key idea, however we do provide examples from other search engines and show what the back-end queries look like. The syntax of search language supported by HotBot includes:

- *Keyword* specifications of the form *searchTerm* requesting all documents that contain the word or the phrase *searchTerm*.
- *Meta words* specifications of the form *keyword:searchTerm* requesting all documents that contain the *searchTerm* in the given *keyword*, e.g., `title:interface` asks for pages that contain the word `interface` in their title.
- *Date meta word* specifications of the form *dateMetaWord:date* requesting all documents created or modified within a specific range of dates, e.g., `within:3/months` asks for pages modified in the last three months while `before:01/01/97` asks for those modified before 1997.
- *Query modifiers* of the form +*searchTerm* or −*searchTerm* requesting all documents that must include (+) or must not include (−) a specified *searchTerm* e.g., `+apple−orange` asks for pages with the word `apple` but without the word `orange`.

- The boolean operators *and*, *or*, and *not* with their obvious meaning.

Following our model of looking at each document as a collection of attribute/value pairs, we now describe how we represent this query language in a configuration file. To represent keyword specifications, we use the `HTML Page` attribute: a given keyword is then specified as the value of this attribute. To represent meta word specifications we use attributes corresponding to the supported keywords: a given search term is then specified as the value of the corresponding attribute. To represent date meta word specifications we use four attributes: the `Modification Date` attribute for constraints on the creation and modification dates of the document and the `Modification Year`/`Modification Month`/`Modification Day` attributes for specifying a range of time for the modification. The following table shows all attribute specifications and how they translate into HotBot's search language.

```
Attribute ::  'HTML Page'            is ''
Attribute ::  'Title'                is 'title:'
Attribute ::  'Domain'               is 'domain:'
Attribute ::  'Link Domain'          is 'linkdomain:'
Attribute ::  'Link Extension'       is 'linkext:'
Attribute ::  'Depth'                is 'depth:'  withtype int
Attribute ::  'Feature'              is 'feature:'  withtype choiceof('title', 'image', ...)
Attribute ::  'Newsgroup'            is 'newsgroup:'
Attribute ::  'Script Language'      is 'scriptlanguage:'
Attribute ::  'Modification Date'    is '' withtype date limitedTo ('is after', 'is before')
Attribute ::  'Modification Year'    is '' withtype int limitedTo ('since year')
Attribute ::  'Modification Month'   is '' withtype int limitedTo ('since month')
Attribute ::  'Modification Day'     is '' withtype int limitedTo ('since day')
```

As you can see, some attributes are declared to accept values of a given type using `withtype`. For example, the `Depth` attribute is specified to be of type `int`, and the `Modification Date` attribute is specified to be of type `date`, two of the types known to the generator tool. The `feature` attribute can only take one of many pre-determined values specified using `choiceof`. Furthermore, we restrict the applicability of attribute operators using `limitedto`. For example, the only attribute operators available for the `Modification Date` attribute are the `is after` and `is before`[1].

To represent query modifiers, we used four attribute operators: `must contain word`, `must contain phrase`, `must not contain word` and `must not contain phrase`. The next table shows all allowable attribute operators along with their translation. The two special variables `$attr` and `val` represent the selected attribute and specified value that are the arguments to the attribute operator.

[1]To be precise, we should have restricted that all attribute accept all operators except the ones reserved for the modification attributes.

| | | |
|---|---|---|
| Attribute Operator :: | ‘must contain word’ | is ‘$attr+$val’ |
| Attribute Operator :: | ‘must contain phrase’ | is ‘$attr+‘‘$val’’’ |
| Attribute Operator :: | ‘must not contain word’ | is ‘$attr-$val’ |
| Attribute Operator :: | ‘must not contain phrase’ | is ‘$attr-‘‘$val’’’ |
| Attribute Operator :: | ‘contains word’ | is ‘$attr$val’ |
| Attribute Operator :: | ‘contains phrase’ | is ‘$attr‘‘$val’’’ |
| Attribute Operator :: | ‘is before’ | is ‘before:$val’ |
| Attribute Operator :: | ‘is after’ | is ‘after:$val’ |
| Attribute Operator :: | ‘since year’ | is ‘within:$val/years’ |
| Attribute Operator :: | ‘since month’ | is ‘within:$val/months’ |
| Attribute Operator :: | ‘since day’ | is ‘within:$val/days’ |

To represent boolean operators we use complex operators. The next table shows the complex operators and their translations using the variables `$lhs` and `$rhs`.

<table>
<tr><td>Complex Operator ::</td><td>‘so long as’</td><td>is ‘$lhs & $rhs’</td></tr>
<tr><td>Complex Operator ::</td><td>‘also’</td><td>is ‘$lhs | $rhs’</td></tr>
<tr><td>Complex Operator ::</td><td>‘except’</td><td>is ‘$lhs ! $rhs’</td></tr>
</table>

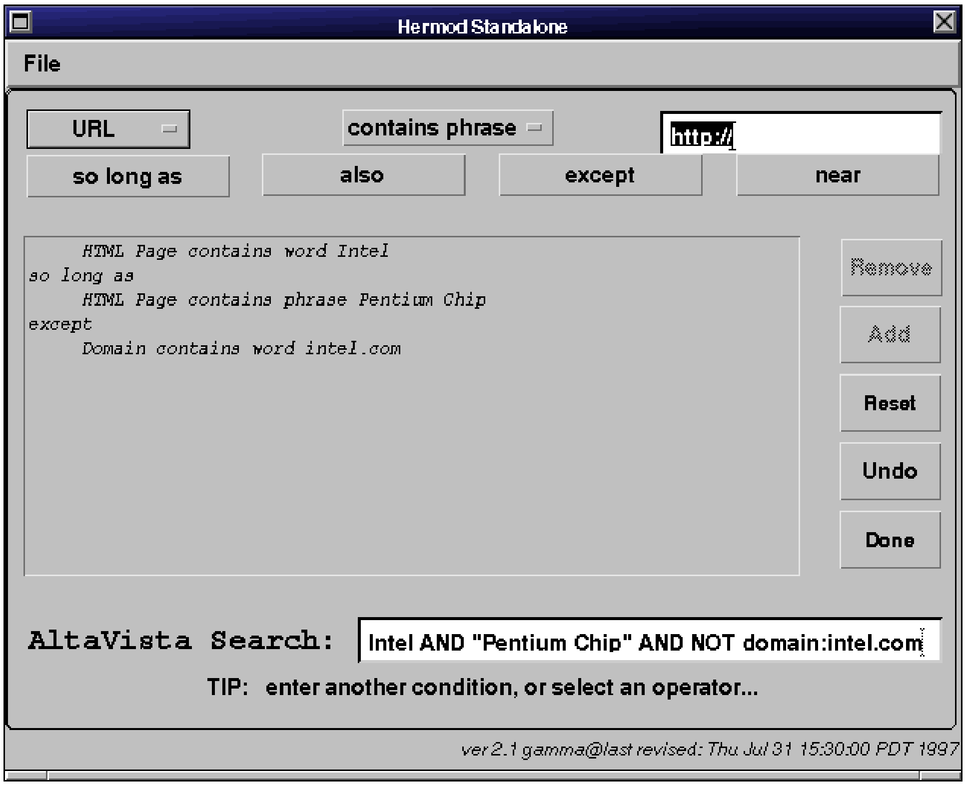


Figure 2: AltaVista query requesting documents on the Intel Pentium Chip.

In Figure 1 we saw a query requesting all documents mentioning both the word `Intel` and the phrase `Pentium Chip` so long as they are not located in the intel.com web site. Figures 2, 3, and 4 show the same

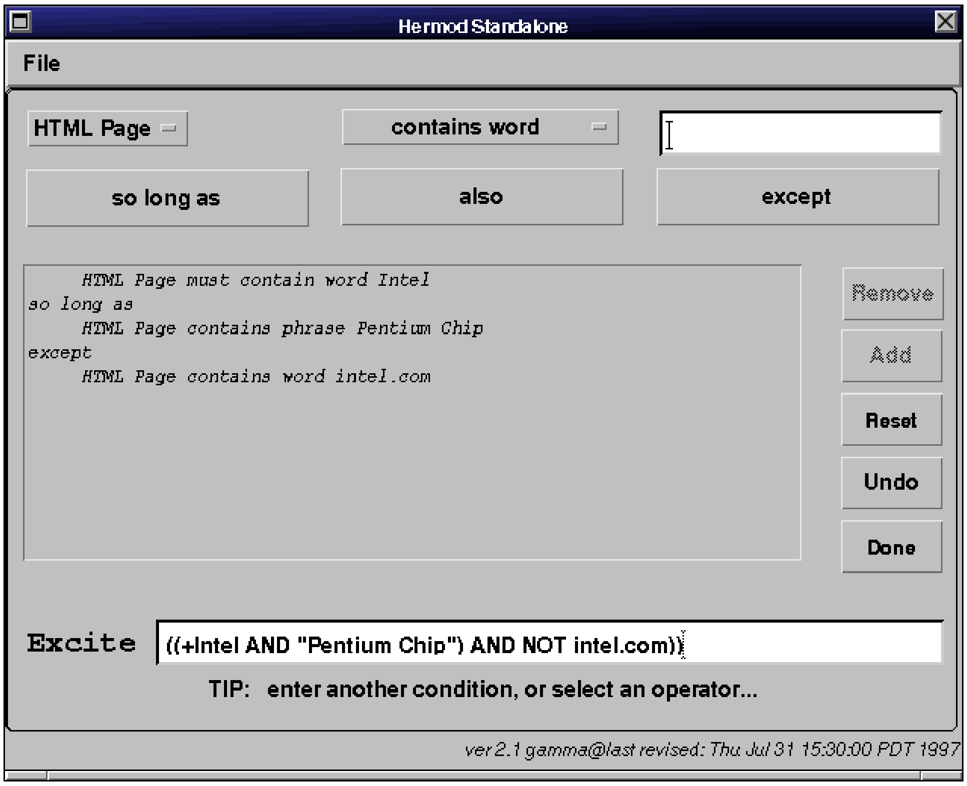


Figure 3: Excite query requesting documents on the Intel Pentium Chip.

query in the interfaces we built for AltaVista, Excite, and Infoseek together with the translation of the query into the different search languages. Figure 5 shows a query requesting all documents that contain the phrase `Bed and Breakfast` as long as they contain a video and have been modified in the last three months. Note that the `feature` attribute can only take one of many values. This screen dump also shows how such restricted values are displayed in a menu that replaces the place holder for the value.

## 6.2 Directories

The second class of applications for which we have used our tool was directories: Four11 [Fou], BigFoot [Big], and LDAP Directories [Howb]. A directory stores information about people and allows people to be retrieved based on their properties. We only describe the generation of the LDAP directory interface. LDAP directories fit very naturally into our data model because they represent people by a collection of attribute/value pairs. The language of access follows the Lightweight Directory Access Protocol Filter Language [Howa] which has the following syntax:

- *attribute operator value* requests people whose value for the specified attribute satisfies the given operator. For example, `mail=*@research.att.com` requests all people whose email address is in the `research.att.com` domain.
- *(&(E1)(E2) ... (En))* requests people for whom all of the conditions *E1*, *E2*, ..., *En* are true.

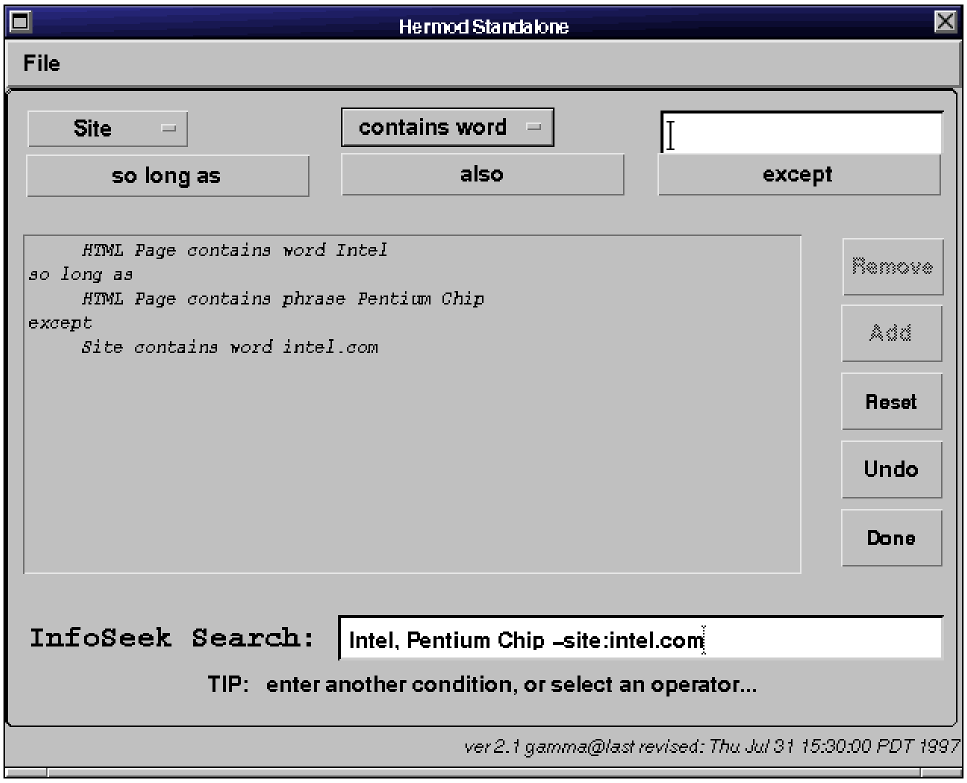


Figure 4: Infoseek query requesting documents on the Intel Pentium Chip

- *(|(E1)(E2) ... (En))* requests people for whom one of the conditions $E1$, $E2$, ..., $En$ is true.
- *(!(E))* requests people for whom the condition $E$ is not true.

To support the expression *attribute operator value* we use atomic queries. For the expressions *(&(E1)(E2)(E3) ... (En))* we use the equivalent form *(&( ... (&(&(E1)(E2))(E3)) ... (En))*. We need to do this, because our current implementation can only support binary complex operators. Similarly for the expression *(|(E1)(E2) ... (En))*. Finally, to support the expression *(!(E))* we have two cases: if $E$ is atomic, then we introduce an attribute operator corresponding to the negation of the attribute operator in $E$. If $E$ is not atomic, then we use the following equivalences:

*(!(&(E1)(E2))) = (|(!(E1))(!(E1)))* and
*(!(|(E1)(E2))) = (&(!(E1))(!(E2)))*.

The attributes supported by the directory server we were accessing were: the first and last name of the person, a boolean attribute describing whether the person is on vacation, and their title, email address, and so on. The LDAP model specifies that we can check whether an attribute equals, contains, starts with, ends with, exists, is similar to, is greater than, or is less than a given value. To support complementation, we introduce additional attribute operators for checking whether any of these conditions is not true. The next two tables contain the configuration file for the LDAP directory interface.

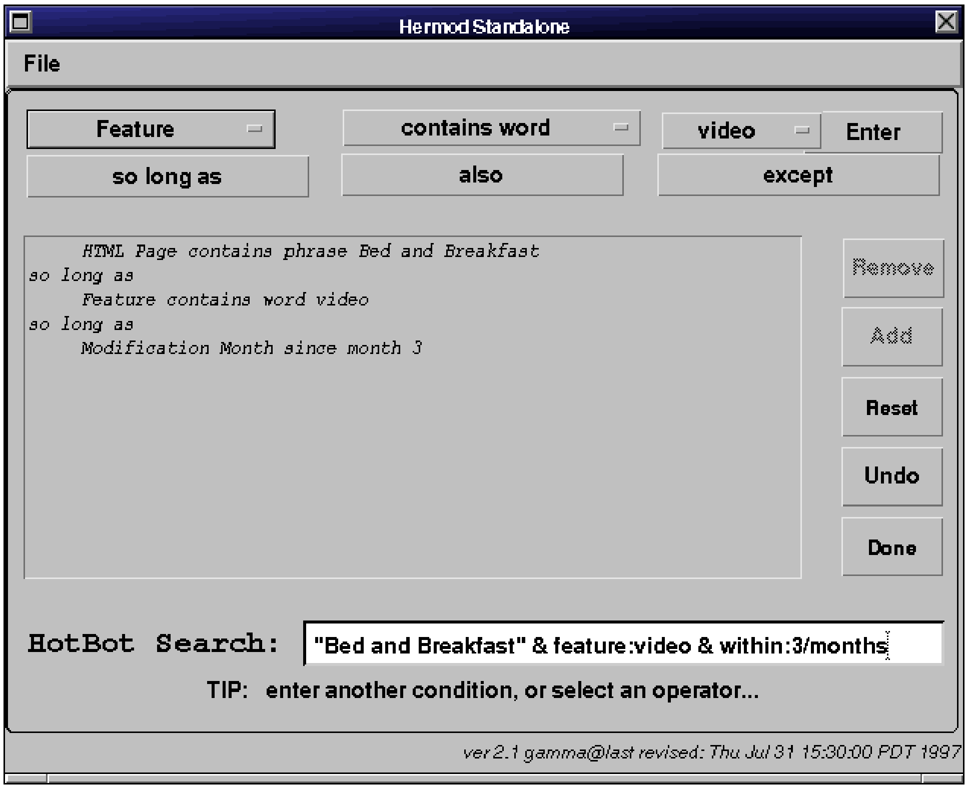


Figure 5: HotBot query requesting documents on Bed and Breakfast.

| | |
|---|---|
| Attribute :: 'first name' | is 'cn' |
| Attribute :: 'last name' | is 'sn' |
| Attribute :: 'on vacation' | is 'onvacation' withtype booleanof('TRUE', 'FALSE') |
| | withvalue 'FALSE' limitedto('is', 'exists') |
| Attribute :: 'title' | is 'title' |
| | withtype choiceof('Employee', 'Manager', ...) |
| | withvalue 'Manager' |
| Attribute :: 'email' | is 'mail' |
| Attribute :: 'organization' | is 'ou' |
| Attribute :: 'home page' | is 'labeledurl' |
| | withvalue 'http://www.research.att.com' |
| Attribute :: 'office phone' | is 'telephonenumber' withtype 'tel' |
| Attribute :: 'office address' | is 'postalAddress' |
| Attribute :: 'home phone' | is 'homephone' withtype 'tel' |
| Attribute :: 'home address' | is 'homepostalAddress' |
| Attribute Operator :: 'is' | is '($attr=$val)' |
| Attribute Operator :: 'contains' | is '($attr=*$val*)' |
| Attribute Operator :: 'begins with' | is '($attr=$val*)' |
| Attribute Operator :: 'ends with' | is '($attr=*$val)' |
| Attribute Operator :: 'exists' | is '($attr=*)' |
| Attribute Operator :: 'is similar to' | is '($attr~=$val)' |
| Attribute Operator :: 'is greater than' | is '($attr>=$val)' |
| Attribute Operator :: 'is less than' | is '($attr<=$val)' |
| Attribute Operator :: 'is not' | is '(!($attr=$val))' |
| Attribute Operator :: 'does not contain' | is '(!($attr=*$val*))' |
| Attribute Operator :: 'does not begin with' | is '(!($attr=$val*))' |
| Attribute Operator :: 'does not end with' | is '(!($attr=*$val))' |
| Attribute Operator :: 'does not exist' | is '(!($attr=*))' |

| | | |
|---|---|---|
| Attribute Operator :: | 'is not similar to' | is '(!($attr~=$val))' |
| Attribute Operator :: | 'is not greater than' | is '(!($attr>=$val))' |
| Attribute Operator :: | 'is not less than' | is '(!($attr<=$val))' |
| Complex Operator :: | 'so long as' | is '(& $lhs $rhs)' |
| Complex Operator :: | 'also' | is '(\| $lhs $rhs)' |
| Complex Operator :: | 'except' | is '(& $lhs (!$rhs))' |

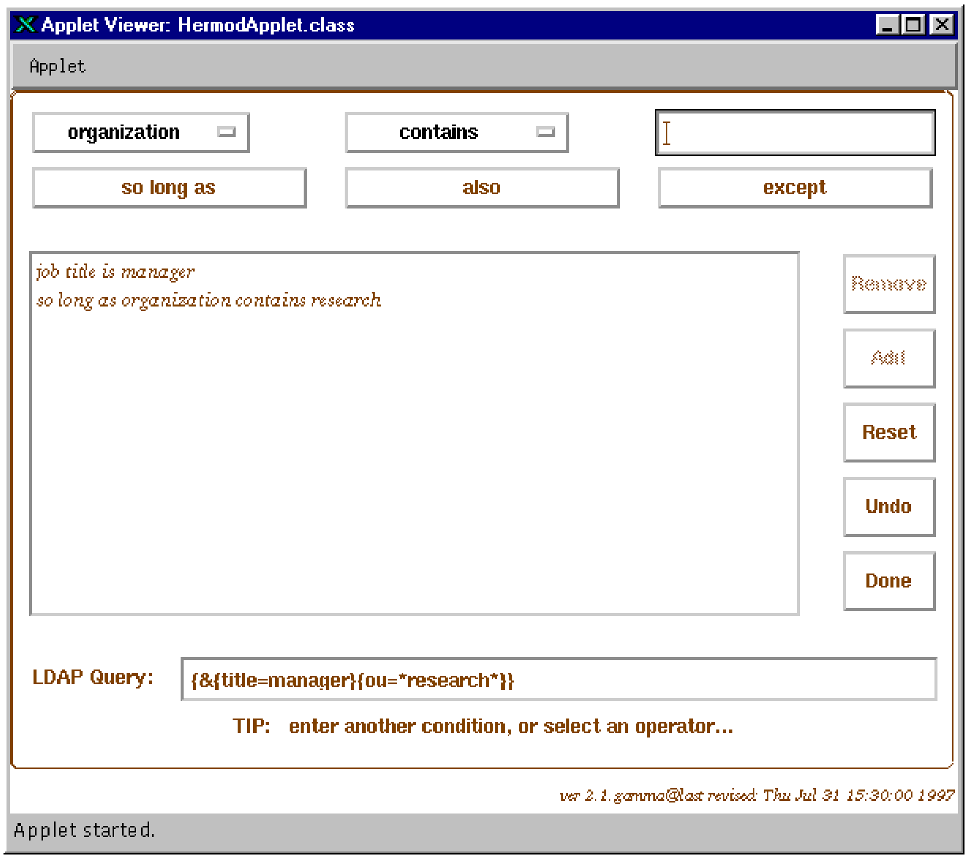


Figure 6: Query Requesting All Managers In The Research Organization From A Directory.

Figure 6 shows a query requesting all people who are managers in an organization whose name contains the word `research`. Figure 7 shows a query from the BigFoot interface requesting all persons whose name is `Tom`, who have an email address in the `aol.com` domain and who have an html page. Figure 8 shows a query from the Four11 interface requesting all persons whose live in `New York`, who work in the `Super Law Firm` and whose name is similar to `Joan`.

## 6.3 Other Information Sources

In this subsection we discuss the configuration of advanced interfaces for domain-specific systems. We have devised interfaces for two systems: the IBM Patent Server [IBM] and the DejaNews usenet search system [Dej]. We only describe the IBM Patent Server whose syntax for advanced search includes:

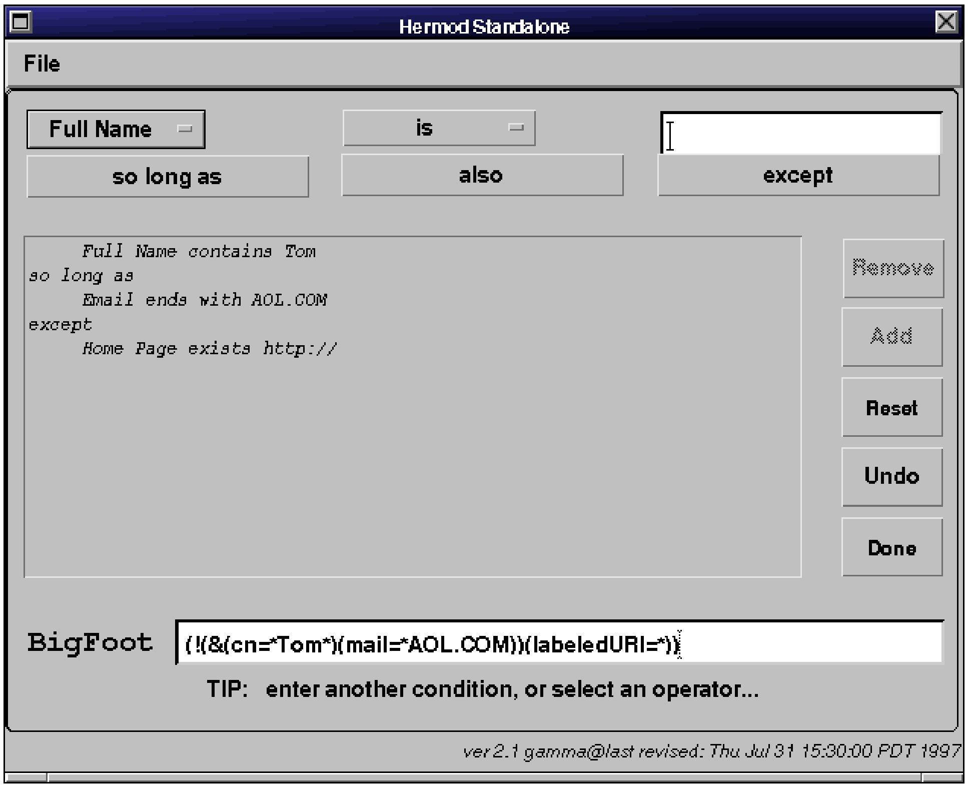


Figure 7: b) A BigFoot Query.

- *search word modifiers* of the form <word> keyword or <thesaurus> keyword. For example, <word> algorithm requests all patents that contain the exact word algorithm while <thesaurus> algorithm requests those that mention any synonym of algorithm.
- *field search operators* of the form word <in> (field1, field2, ...) with the meaning that a specified *word* must appear in the given field names. For example, algorithm <in> title requests the patents containing the word algorithm in the title. The following fields are supported by the query language: *title*, *abstract*, *assignee*, *inventor*, *agent*, *claims*, *otherrefs*, *usrefs*, *names*, and *summary*.
- the *proximity operators* <near>, <paragraph>, <sentence>, and <order>. For example, algorithm <near> graph requests patents with words algorithm and graph in near proximity, algorithm <sentence> graph requests them in the same sentence, algorithm <paragraph> graph requests them in the same paragraph and algorithm <order> graph requests them in the specified order.
- the *logical operators* <and>, <or>, <not> and <accrue>. For example algorithm <and> graph requests patents that contain both words algorithm and graph.

To represent search word modifiers, we use the attribute patent, the attribute operator contains exact word or contains synonym and the value keyword. To represent field search operators of the form word <in> (field1 field2 ...) we use the equivalent form of (word <in> field1) <and> (word <in> field2) <and> .... Each word <in> field is represented by an attribute for the field and the operator contains. To represent proximity operators, we use the attribute patent, the attribute operator contains and the word

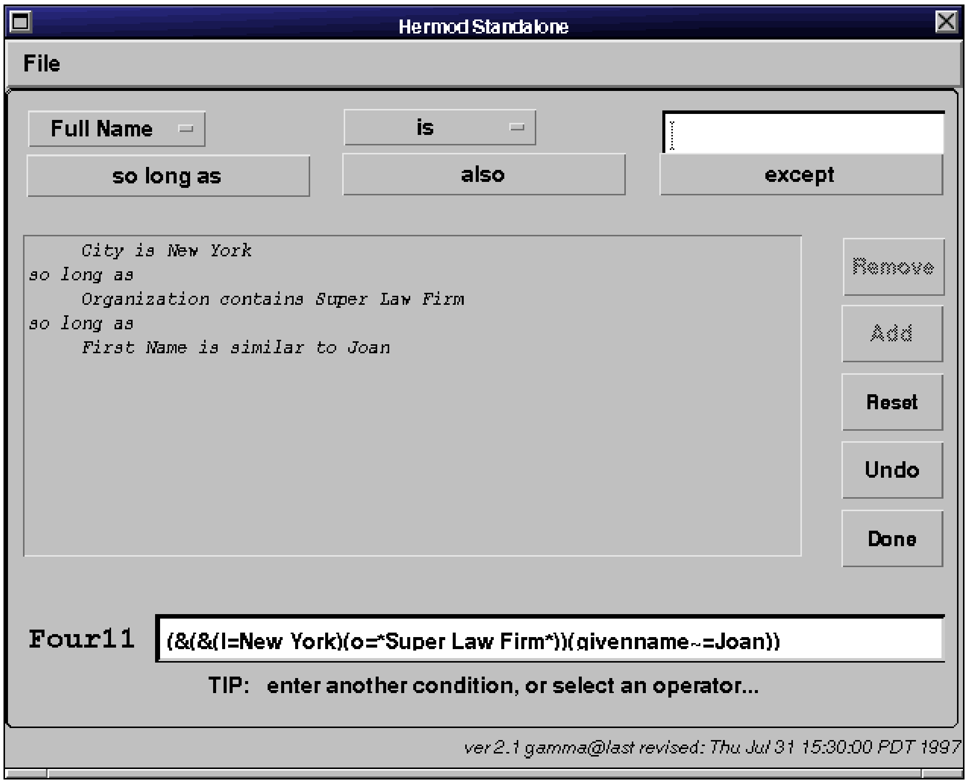


Figure 8: A Four11 Query.

as a value. Proximity operators themselves are represented as complex operators. Thus, `algorithm <near> graph` is represented as `patent contains algorithms near patent contains graph`. Logical operators are represented using complex operators, as usual.

Figure 9 shows a query from IBM patent server interface requesting all patents on databases, whose inventor's name ends with Bell and whose assignee contains the word AT&T. Figure 10 shows a query from the DejaNews interface requesting all articles posted to the `comp.lang` newsgroup, on a subject containing `Tcl/Tk`, posted the first day of the year, unless it is an article about `Java`.

# 7 Discussion

We have demonstrated the utility of using a generator tool to construct advanced visual query interfaces for a variety of information sources available on the Web. Our generator tool can be very flexibly configured to support a variety of selection-based query languages and to construct interfaces with a uniform look and feel. Our examples demonstrate that we can support consistent interfaces very easily by specifying a simple user configuration file.

Traditionally, user interfaces are built using programming languages. Special interface builders, such as Visual Basic (Basic), Delphi (Pascal), SpecTcl (Tcl/Tk), wxPython (Python), NetFustion Objects (HTML) and Visual Cafe (Java), are popular graphical interface builders. These tools emphasize the building of the visual layout of the interface not the data manipulation aspect of the interfaces. Query translation to a

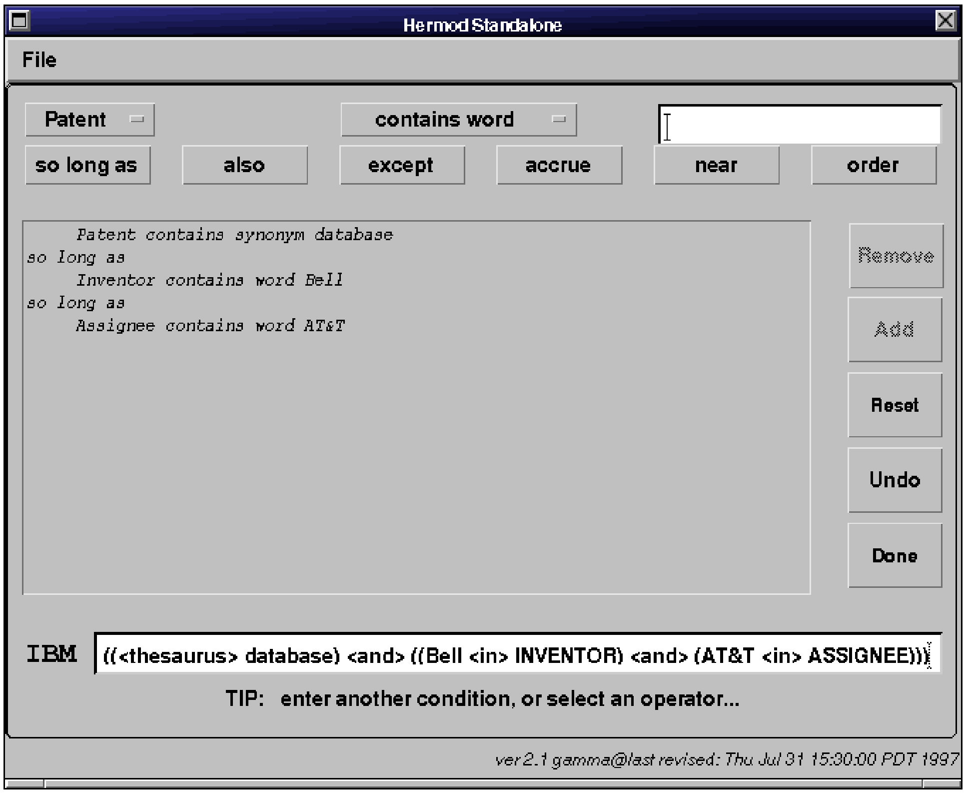


Figure 9: An IBM Query

back-end query system still needs to be supported programmatically. The innovation of our tool is the fact that it supports declaratively the customization of the language interactions from the generated interface to the underlying system.

Krishnamurthy and Zloof [KZ95] have studied the problem of generating interfaces by declarative specifications. Their emphasis is on data visualization interfaces for data computed by database queries which allow subsequent interactions with the presented data. The GraphLog language [CM90] allows the specification of database queries graphically while supporting the translation from the visual queries to the queries of a back-end database system [CMV94]. Their emphasis is on query visualization for general database queries. A more relevant work is the one by Phanouriou and Abrams [PA97] who present the Javamatic system which can generate a graphical user interface, then invokes commands to a legacy system transparently to the user. Their system uses a drag-and-drop technique to generate the graphical user interface and can only be used for command-line legacy systems.

# References

[Alt] AltaVista. The AltaVista Search Engine. Available at http://www.altavista.digital.com.

[Big] BigFoot. The BigFoot Directory. Available at http://www.bigfoot.com.

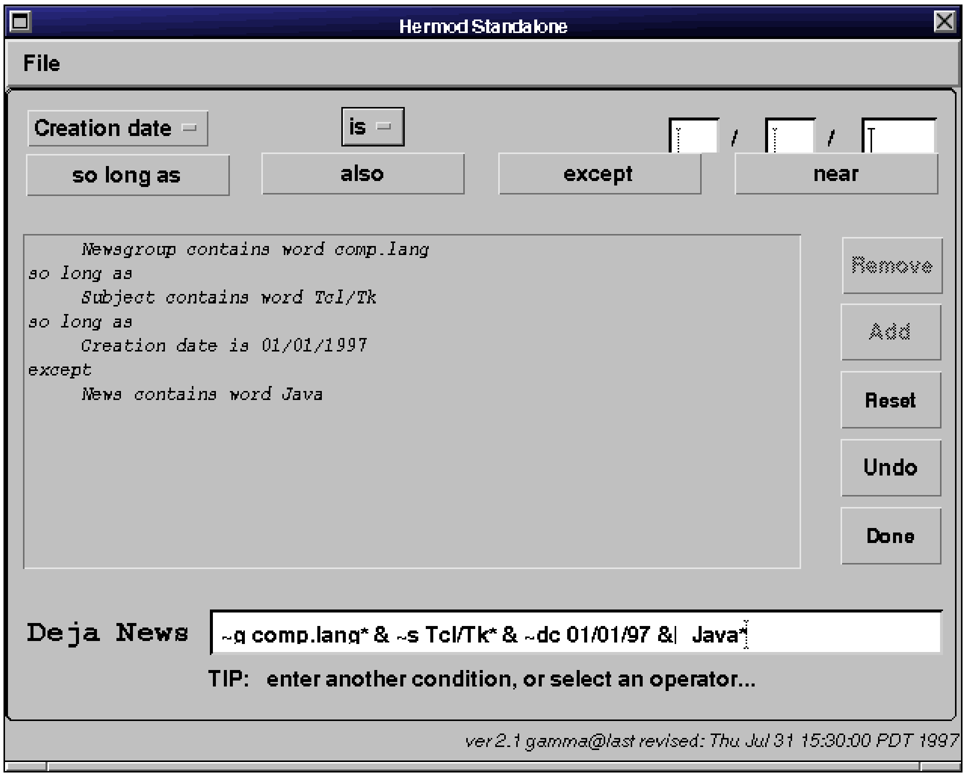


Figure 10: A DejaNews Query.


[CM90] M.P. Consens and A.O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CMV94] M.P. Consens, A.O. Mendelzon, and D. Vista. Deductive Database Support for Data Visualization. In *Proceedings of the 4th International Conference on Extending Database Technology*, pages 45–58, 1994.

[Dej] DejaNews. The DejaNews Server. Available at http://www.dejanews.com.

[Exc] Excite. The Excite Search Engine. Available at http://www.excite.com.

[Fou] Four11. The Four11 Directory. Available at http://www.four11.com.

[Hot] HotBot. The HotBot Search Engine. Available at http://www.hotbot.com.

[Howa] T. Howes. A String Representation of LDAP Search Filters. Network Working Group Request for Comments 1558. Available from http://www.pmg.lcs.mit.edu/cgi-bin/rfc/view?number=1558.

[Howb] T. Howes. Lightweight Directory Access Protocol. Network Working Group Request for Comments 1777. Available from http://www.pmg.lcs.mit.edu/cgi-bin/rfc/view?number=1777.

[IBM] IBM. The IBM Patent Server. Available at http://patent.womplex.ibm.com.

[Inf] Infoseek. The Infoseek Search Engine. Available at http://www.infoseek.com.

[KZ95] R. Krishnamurthy and M. Zloof. RBE: Rendering By Example. In *Proceedings of the 11th International Conference on Data Engineering*, pages 288–297, 1995.

[PA97] C. Phanouriou and M. Abrams. Transforming Command-Line Driven Systems to Web Applications. In *Proceedings of the 6th International World Wide Web Conference*, 1997.

[Sal89] G. Salton. *Automatic Text Processing: the Transformation, Analysis and Retrieval of Information by Computer*. Addison Wesley, 1989.