An Extensible Constructor Tool for the Rapid, Interactive Design of Query Synthesizers

Michelle Baldonado, Seth Katz, Andreas Paepcke,
Chen-Chuan K. Chang, Hector Garcia-Molina,Terry Winograd
Gates Building 4A
Stanford University
Stanford, CA 94305
E-mail: {michelle, sethkatz, paepcke, kevin, hector, winograd}@cs.stanford.edu

ABSTRACT
We describe an extensible *constructor tool* that helps information experts (e.g., librarians) create specialized *query synthesizers* for heterogeneous digital-library environments. A query synthesizer provides a graphical user interface in which a digital-library patron can specify a high-level, fielded, multi-source query. Furthermore, a query synthesizer interacts with a query translator and an attribute translator to transform high-level queries into sets of source-specific queries. We discuss how the constructor can facilitate discovery of available attributes (e.g., title), collation of schemas from different sources, selection of input widgets for a synthesizer (e.g., a text box or a drop-down list widget to support input of controlled vocabulary),, and other design aspects. We also describe a prototype constructor we implemented, based on the Stanford InfoBus and metadata architecture.

KEYWORDS: constructor tool, query synthesizer, regional schema, query generation, query translation, attribute translation, metadata architecture, schema

1. INTRODUCTION

With the advent of large, rapidly evolving heterogeneous digital libraries, *patrons* are faced with several difficulties when trying to submit a query to multiple sources. First, the patron must identify the right sources to use. Second, the patron must determine what queries to submit to the sources. While some sources can be queried with a simple list of keywords, many sources require the specification of fields to search or operators to use (e.g., for proximity searching). Thus, the patron must determine the appropriate fields and operators to use for each source. Furthermore, the patron needs to know what values to use in the queries. For example, a given field may require its values to come from a controlled vocabulary (e.g., JOURNAL must be one of CACM, TODS, TOIS), certain keywords may be preferable (e.g., automobile over car), or values should be of a given type (e.g., integers, not strings).

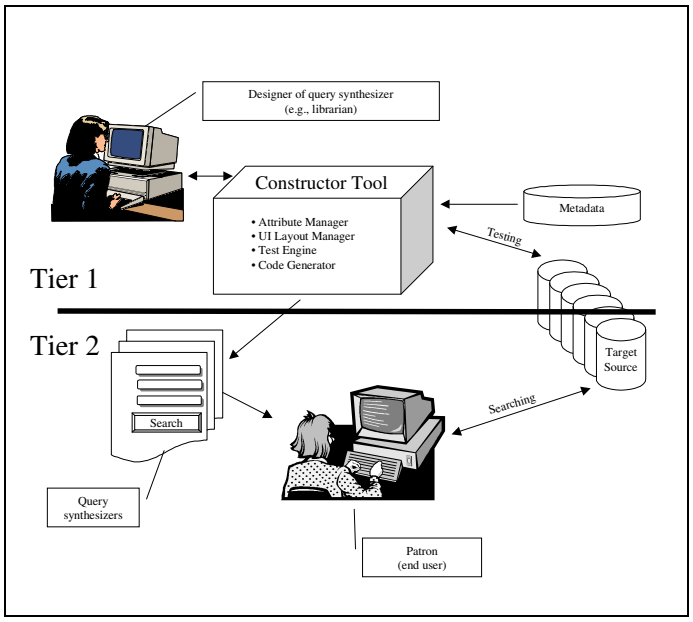

Figure 1: Two-tier approach to query formulation

In this paper, we propose a two-tier approach to facilitating query formulation in evolving, heterogeneous digital-library environments (see Figure 1). The first tier in our approach revolves around a *constructor tool* that is used periodically by an expert *designer* (e.g., a librarian) to explore the currently available information sources and their idiosyncrasies. By using the constructor tool, the designer produces one or more *query synthesizers* for specific *tasks* or domains. These synthesizers form the basis of the second tier. A query synthesizer provides a graphical user interface (see Figure 3) in which a digital-library patron (end user) can specify both a high-level, fielded query and a set of diverse target sources for that query. Furthermore, a query synthesizer interacts with a *query translator* and an *attribute translator* to transform high-level queries into sets of source-

specific queries. Our two-tier approach allows an information expert to codify knowledge about tasks and

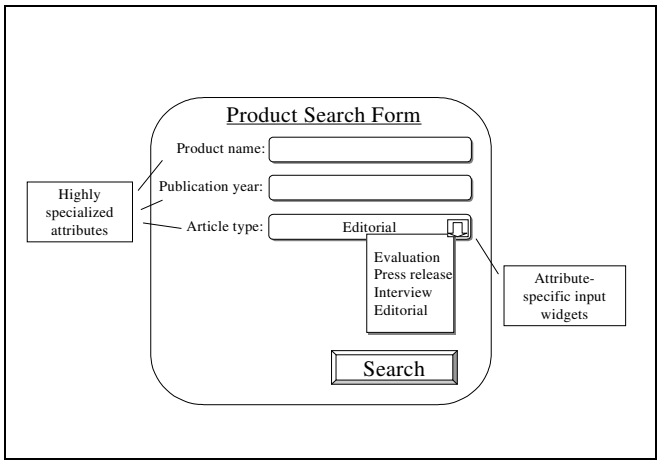

Figure 2: A simple GUI provided by a query synthesizer for product search

sources ahead of time for the patron. As a result, patrons can more easily generate sophisticated queries.

The context for our approach is the Stanford Digital Library project, which provides uniform access via the InfoBus [7] to a heterogeneous set of information services, search services, and metadata services. The InfoBus includes a set of protocols in a CORBA-compliant distributed object architecture that allows services and clients to communicate via remote method calls. Figure 2 presents an overview of the Stanford Digital Library architecture, including the constructor tool and its elements.

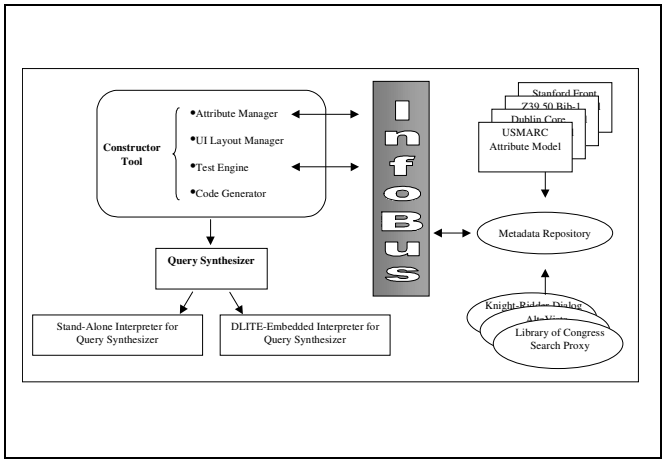

Figure 3: Stanford Digital Library Architecture, Including the Constructor Tool

Very briefly, the constructor and synthesizers are used as follows. The designer first establishes the task for which patrons will use the new synthesizer. He uses the constructor to look for appropriate information sources and learn what fields and operators are available for searching. The constructor obtains this information by contacting sources and metadata repositories through the InfoBus. The designer next selects the inputs that the patron will be allowed to enter, their formats, and how these inputs will be incorporated into the final query. The designer can also "test run" queries to ensure the design is correct. Finally, when the patron uses the resulting synthesizer, he simply fills out the form, without having to understand how and what sources are contacted.

Several challenges arise in designing such a constructor tool that supports the rapid, interactive design of query synthesizers. In this paper, we identify the following design issues, outline our approach to each issue, and discuss the current state of our constructor-tool implementation with respect to each issue.

- Schema access. How does the designer determine what fields are available for search at all the sources relevant to the synthesizer to be created?
- Schema collation. How does the designer reconcile schemas if the synthesizer will search multiple sources at the same time?
- Constructor-tool user interface. How does the designer interact with the tool to produce a synthesizer?
- Architecture and implementation. How was our prototype constructor tool built?

2. SCHEMA ACCESS

After identifying sources relevant to a target query synthesizer, the designer needs to examine the schemas of those sources. A source's schema is the collection of fields available for searching and retrieving portions of documents. By understanding the source schemas, the designer can further gauge each source's relevance, and can ensure that the new synthesizer provides a maximum of support in using fielded queries. The use of fields in queries is important, because it can significantly improve search results for sources that maintain indexes. For example, a database of online computer trade magazines will yield thousands of results to a query for the term "notebook computer." In contrast, the result set is limited to a meaningful couple of dozens when the query specifies that the result documents are to be of type 'evaluation' and that the publication year should be 1997. While many sources currently on the World-Wide Web do not support such fielded requests, many high-quality sources such as the Dialog Information Service have done so for years. Some Web sites (e.g., specialized electronic-mail address finders) are beginning to introduce attribute-based search as well.

Finding out which attributes may be used with any given source raises the problem of schema access: How can the schemas be inspected and compared? Some commercial information providers support such metadata browsing. Others do not. Even when such browsing facilities are available, the issue of differing attribute naming conventions

remains. Many attribute-naming schemes have been developed for full-text sources, notably the Library of Congress' MARC scheme [10], Z39.50's BIB1 [6], or, more recently, Dublin Core [9]. The schema access problem would be easier to solve if target sources would all support one or several such naming schemes in their entirety. Unfortunately, this is not the case. Many target sources present entirely non-standard attributes or they support only a subset of the standard sets. This may be because the sources contain very specialized contents or because their content is indexed only on a few of the attributes.

Even if target sources adhered to more orderly schemes, attribute names alone are not enough. More detailed information about each attribute is needed. The most obvious is the data type of the allowable attribute values. Furthermore, it is important to know which operators are appropriate for an attribute. A good query synthesizer should, for example, warn patrons if they try to use truncation ('wildcards') in a numeric field, unless the underlying search engine can support this. Similarly, some search engines allow keyword searching over some fields, but only 'whole-string' (phrase) searching over others. Complete solutions to the schema access problem must include this kind of operational information as well.

Relational databases have long supported schema access through data dictionary modules. They allow users or applications to explore which relations exist in the database and which attributes comprise each relation. For relational databases, this job is somewhat easier than for text retrieval, because the organization of data in relational systems is much more structured and well defined.

In our construction tool prototype, we have addressed the problem of schema access by using our comprehensive metadata architecture that allows for the cataloguing, browsing, searching, and translation of metadata [1]. The right-side portion of Figure 3 summarizes these metadata access facilities of the prototype. Two aspects of the architecture are relevant to the schema access problem: attribute models and source-specific metadata that is available from our search proxy objects. An attribute model is a machine-accessible representation of a coherent field convention. For example, one of our attribute models describes the Dublin Core naming scheme. Each Dublin Core field is represented by a programming object that contains all of the information about that field. We can search over these objects, and can find out, for example, which attributes are related to 'authorship', or we can extract an explanatory description of a given attribute. Our synthesizer constructor uses attribute models to display relevant information for the synthesizer designer.

Attribute models are independent of any particular source. In order to find out which subsets of attribute models are supported at a given source, our constructor tool turns to the corresponding library search proxy (LSP), shown at the bottom right of Figure 3. An LSP is a wrapper that represents an information source. Each LSP provides a standard method that returns the schema of the source. That schema includes all the attributes actually supported, as well as any local restrictions, such as usability with query language operators.

The metadata repository in Figure 3 provides all of the search proxies' metadata in one place. The constructor tool queries the metadata repository whenever it needs to learn about attributes supported by any given search proxy. As the query-synthesizer designer adds more target sources, the constructor can thereby provide feedback about which attributes are common to the sources.

By architecting our constructor tool so that it interacts with the Stanford metadata architecture, we ensure that it is extensible. As new sources and attribute models are added to the InfoBus, they will be dynamically available to the constructor tool.

3. SCHEMA COLLATION

If a query synthesizer is destined to be used with a single source only, schema access facilities suffice in helping synthesizer designers find the correct attributes to use. Otherwise, the schemas of potential target sources somehow need to be reconciled. For example, the designer or underlying translation facilities must determine what fields are analogous at each source and can be searched in a joint fashion across sources. Some fields may not have an equivalent at all target sources, and the designer must decide whether to include them in the synthesizer and how. The designer must also specify constraints on the values patrons may enter for each field. The runtime system for the synthesizers must then enforce these constraints once the synthesizer is deployed. Finally, the designer must specify a strategy for merging results from different sources and ranking them for the patron.

In this section, we explore various possible approaches to schema collation. Several of these approaches have been developed by the database community over the years. See, for example, [2] for a survey. We will explain that the solution we chose for schema collation is primarily motivated by the heuristic nature of Digital Library usage, which tends to require a less rigorous, but more flexible approach to this problem than is common in database systems.

Refer to Figure 4 for a graphical summary of Options 1-5 below.

Option 1: One approach is to develop *a priori* a single, unified, **global schema** that is used for query formulation. Pre-computed mappings are used to transform global queries into equivalent queries for each target source. An example of this approach can be found in [8]. It is illustrated in the left-most portion of Figure 4. Attribute $\boldsymbol{a}$, which is common to source schemas S_A, S_B, and S_C, is mapped to attribute $\boldsymbol{w}$ in the global schema. Attributes $\boldsymbol{b_1}$, $\boldsymbol{b_2}$, and $\boldsymbol{b_3}$, which are similar, are combined and mapped to global attribute $\boldsymbol{x}$.

Similarly for c_1,$c_2 \rightarrow y$, and $d \rightarrow z$. In order for the global attributes to be understood at the target sources (bottom of Figure 4), they need to be translated.

One advantage of this approach is that it allows for the removal of purely syntactic differences. For example, one source might call the required payment for an item 'cost', while another calls it 'price'. A global schema can help users by unifying such gratuitous differences. A disadvantage of the approach is that global schemas need to be revised

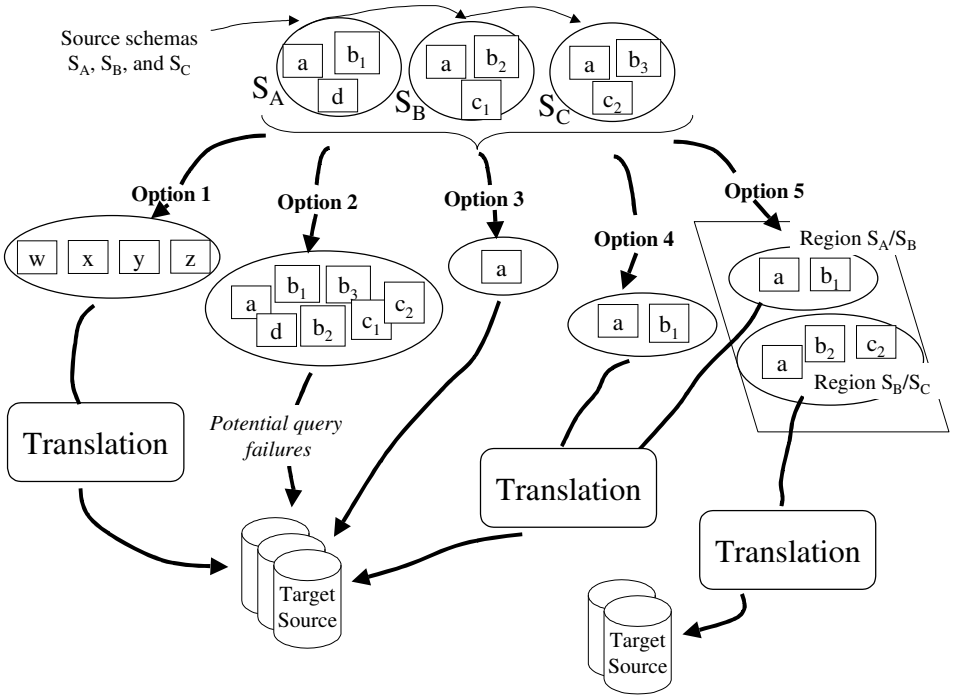

Figure 4: Different solutions for schema collation. Elements a, b_1, etc. are attributes. Elements w, x, y, and z are global schema attributes. Subscripted attributes are similar and can be translated into each other (e.g., b_1, b_2).

whenever new sources join the set of targets. For example, consider two sources describing items for sale. If one uses 'product number' while the other uses 'serial number' as an identifier for each product, a global schema might neatly unify the two by using an attribute called 'product identifier'. If a new source is added that records both a 'serial number' and a 'product ID', then the global schema needs to introduce a second product ID attribute.

Another disadvantage is that sometimes specialized attributes supported only by some sources are not available at the global level at all, because they cannot be mapped to other sources and can therefore not be accommodated in the global schema. Construction of effective global schemas also usually requires careful, manual labor.

Option 2: Another approach to schema collation is to take the **union of all attributes**, and to present them all to the synthesizer designer. The advantage is that this is straightforward computationally, and no attributes are 'abstracted away' and made inaccessible. An obvious problem is that the number of attributes can be very large, potentially overwhelming the designer. Another problem is that the designer must consider which of the sources support each attribute. If all attributes are used indiscriminately, incompatibilities will lead to query failures because not all attributes of the union are supported at all sources.

Option 3: A third approach is to use the **intersection of target schemas**: only those attributes that are supported by all target sources of interest are presented to the synthesizer designer as viable attributes to include in the query synthesizer. The advantages include both ease of computation and reduction of the number of attributes presented to the designer. A disadvantage is that a single source with a very unusual or small set of attributes can drain the intersection of most or all attributes.

Option 4: The intersection approach can be improved upon by adding partial attribute translation facilities. Through this approach, the intersection of attributes is enlarged. The better the translation facilities, the more attributes can be used across a larger number of sources. For example, in Figure 4, the intersection (attribute a), is enriched by attribute b_1 which is then translated to b_2 and b_3 where appropriate.

Various translation techniques can be employed. For example, attributes that are contributed by all sources but differ in name for each source would normally be excluded in the intersection solution. They can be included if their semantic equivalence can be recognized. In this case, a single attribute can be retained in the intersection as a placeholder. As in the global schema approach, the query translation machinery then provides the proper mappings when queries are generated from the synthesizer and are submitted to the various target sources. This regains some of the comprehensiveness of the union approach, while maintaining the desirable tightness of the intersection solution.

Similarly, if the value types of corresponding attributes in multiple sources differ, then attribute value translation can be used to provide the schema uniformity necessary to keep the intersection large enough for practical use. For example, if an attribute in one schema calls for an array of integers representing the coordinates of a place on a map, and a corresponding attribute in another source calls for a string containing the same information in another coordinate system, then a synthesizer can enforce input of one or the other format, with attribute value translation taking care of the necessary adjustment.

Finally, controlled query degradation can be used to enlarge attribute intersections. For example, suppose the 'abstract' attribute is supported by some of the target sources, but not by others. If the problem sources support an 'anywhere' attribute that causes searches to range over the entire record, then any occurrence of the 'abstract' attribute can be replaced by 'anywhere' during the final query translation process. Less drastically, if a target source supports 'body', then occurrences of 'abstract' can be generalized by using 'body'. This transformation would qualify documents that contain the desired keywords in the main body, not necessarily in the abstract. The transformation will degrade the query because precision is decreased, but the query will still run over all the sources. We have frequently found that it is preferable to trade some loss of precision for uncomplicated query applicability to multiple sources. This is especially true if patrons are supported in analyzing large result sets through ranking, clustering, and other exploratory tools. We have discussed the relevant tradeoffs and limitations of this particular transformation technique elsewhere [3].

In the general case, discovery of semantic equivalence of attributes is very difficult to automate and is tedious to accomplish manually. But when there is a relatively small number of attribute models, such analysis can be performed by hand, at least for some of the major models, and perhaps concentrating on particularly important attributes within those models.

Option 5: Our query synthesizers are really a variation of Option 4, in that they use attribute translation during schema collation. In contrast to Option 4, they do not try to unify large areas of schemas at the same time. Instead, they represent small, **regional schemas**. These schemas differ from global schemas in that they are not comprehensive, but contain only a small set of attributes that are of interest for a particular task. They are also different from global schemas and the intersection approach in that they need not reliably cover all possible target sources. Patrons use the synthesizers and submit the resulting queries to various sources, some of which might not have been anticipated by the synthesizer designer. Sometimes this will work well, thanks to the techniques described for Option 4 above, other times it may not. We are finding that as more patrons become accustomed to Web search engines, they understand the fact that information retrieval is often heuristic, and that the possibility of failure may include the inability of some sources to perform optimally, or even properly for all queries. Rather than taking the all-or-nothing approach of global schemas, or the very conservative approach of schema intersection, our regional schema approach, coupled with some attribute translation, attempts to expose patrons to more sources without unduly burdening system administrators with schema maintenance. The right-most portion of Figure 4 shows two schema regions. One is composed from schemas S_A and S_B, the other from S_B and S_C. Different translation facilities do their best to make each region usable with as large a family of target sources as possible.

For the implementation of our construction tool prototype we initially decided to use Option 1 in our schema collation implementation. In particular, we used a subset of USMARC as our global schema. USMARC is widely used in libraries and it covers a broad range of library-related metadata needs beyond the naming of standard document attributes. For example, it provides for attributes that store the physical location of an item, its price, and physical format. Many of these attributes can be generalized and reused in a digital setting like ours. The format attribute, for example, could be used to record whether a document is RTF, Postscript, or some other electronic format. For our initial explorations, USMARC proved to be a rich source of metadata attributes for our digital library setting. Using USMARC throughout the system made the creation of query synthesizers easier, because the USMARC attribute definitions provided a 'lingua franca' of catalog-related metadata.

Eventually, however, we felt that we were stretching the analogy between physical and digital libraries too far. This became most obvious as we were creating collections of online items that were not 'documents' in a traditional sense. For example, we needed to manage document payment through online subscription facilities. Patron accounts were modeled as items in subscription collections. We wanted to search over these account collections in the same way we searched over a bibliographic data source. Other examples were patron profiles and access right records. This broadening of the collection notion arises from the technical realization that collections of electronic books can be managed with similar underlying technology as collections of payment accounts, access rights, or patron profiles. All of

these share a need for base facilities such as persistence, transaction support, indexing, clustering, and searching. This structural unification of administrative and content information in digital library systems is technically economical. Beyond this technical argument, its conceptual uniformity simplifies the construction of unified interfaces for a broad range of digital-library activities.

However, USMARC cannot reasonably be stretched to cover such a diversity of metadata needs, and changing standards is a very difficult process. Even if this were not the case, the modularity inherent in the regional schema approach was preferable, given our wide spectrum of attribute usage. We therefore opted for Option 5. By combining our translation facilities with a carefully designed metadata browser, our solution is to allow information experts to create synthesizers containing attributes from either the intersection or the union of a set of attribute models. As will be shown later, the browser warns the synthesizer designer when attributes are weakly supported for a particular set of sources. Since our attribute translation machinery often provides graceful degradation of query processing in the face of unsupported attributes, the designer may decide to include attributes in a synthesizer even though they are not supported by all the sources for which the synthesizer is intended.

The following section describes how the user interface of our synthesizer construction tool helps designers construct regional schemas and corresponding input forms.

4. CONSTRUCTOR TOOL USER INTERFACE

Figure 5 illustrates how we implemented schema collation Option 5 in our prototype. A synthesizer designer uses our constructor tool to choose sources and attributes. The designer selects sources and attribute models from the respective pull-down menus (top of Figure 5). Each such selection causes that source or model to be added to a table (the Sources/Models table of Figure 5). The designer can use this table to explore the relationship between sources and attribute models. Each cell in the table describes how many attributes from the associated attribute model are supported by the associated source. For example, the table shows that the Yahoo source supports four StanfordFrontModel attributes, while Folio, Stanford University's library catalog supports all of that model's attributes.

The table in Figure 5 provides a compact overview of the relationships between sources and attribute models. The designer can gain a more in-depth understanding by probing the relationships between sources and individual attributes. Figure 5 also shows that an attribute model has been "opened" to reveal its member attributes: The entries below 'StanfordFrontModel' show attributes such as 'Title' and 'Abstract'. In contrast, the 'DublinCoreModel' above is closed, and only shows the number of supported attributes. Each attribute-level cell of the opened model reveals whether or not the attribute is supported, while the attribute-model-level cells continue to reveal summary information about the model. For example, The designer can see that Lycos supports 'Abstract', but not 'PublicationYear', while Folio does support the 'PublicationYear' attribute.

Selecting an attribute (by checking its associated check box) causes the attribute to appear in the lower panel of Figure 5. The order in which the attributes appear in this lower panel corresponds to the order they will appear in the generated query synthesizer. Accordingly, the tool provides buttons that can be used to edit this order.

After deciding what attributes should be included in the synthesizer, the designer must decide how the synthesizer should interact with the user in order to formulate a query. Important questions to consider are how the user enters the appropriate information and how these inputs will be incorporated into the final query that is sent to the source. Note that this involves much more than simply designing the form the patron will see (Figure 2 shows a stylized sample form seen by the patron). Based on information obtained through browsing metadata, the designer might decide that the synthesizer GUI should have pull-down menus listing valid values for controlled vocabularies, or that it should enforce the required data type. Furthermore, the designer might decide to allow patrons to enter values that will have to be translated into the format requested by the source (perhaps the source requires value formats that the patron is unlikely to use). This means the designer must specify the value translation requirements. Finally, the designer must provide a mapping from the terms used on the input form to facilities available in an underlying query language.

Requirements for input widgets to use in the resulting synthesizers depend in large part on the medium of the information being searched. For example, image databases will tend to require more sophisticated input devices to afford pointing, drawing bounding regions, or specifying color and shapes. Purely textual target sources are more easily satisfied with text-based form interfaces.

Another factor that enters into the complexity of the interface design for the synthesizers is the structure of the data being searched. If the data is of regular structure, then very specialized interface mechanisms may be used to facilitate search over that data. For example, Query By Example [11] is a query-input interface for relational databases. It presents the user with a template of a relation. Users (patrons in our context) insert search terms into the correct fields. The search engine takes the resulting partially filled relation as a prototype and uses it to produce result tuples. There are other ways in which data structure can be used for the design of highly specialized query synthesizers. The interface to a target source about cars, for example, could show the image of a car, and could allow patrons to point to the parts they wanted more information on. Similarly, group photos can be used to let patrons extract information about sets of people. In short, synthesizer constructors need to be highly extensible to allow the addition of new input widgets over time. If the run-time query synthesizer interpreter will be embedded in a Java virtual machine, generality of the

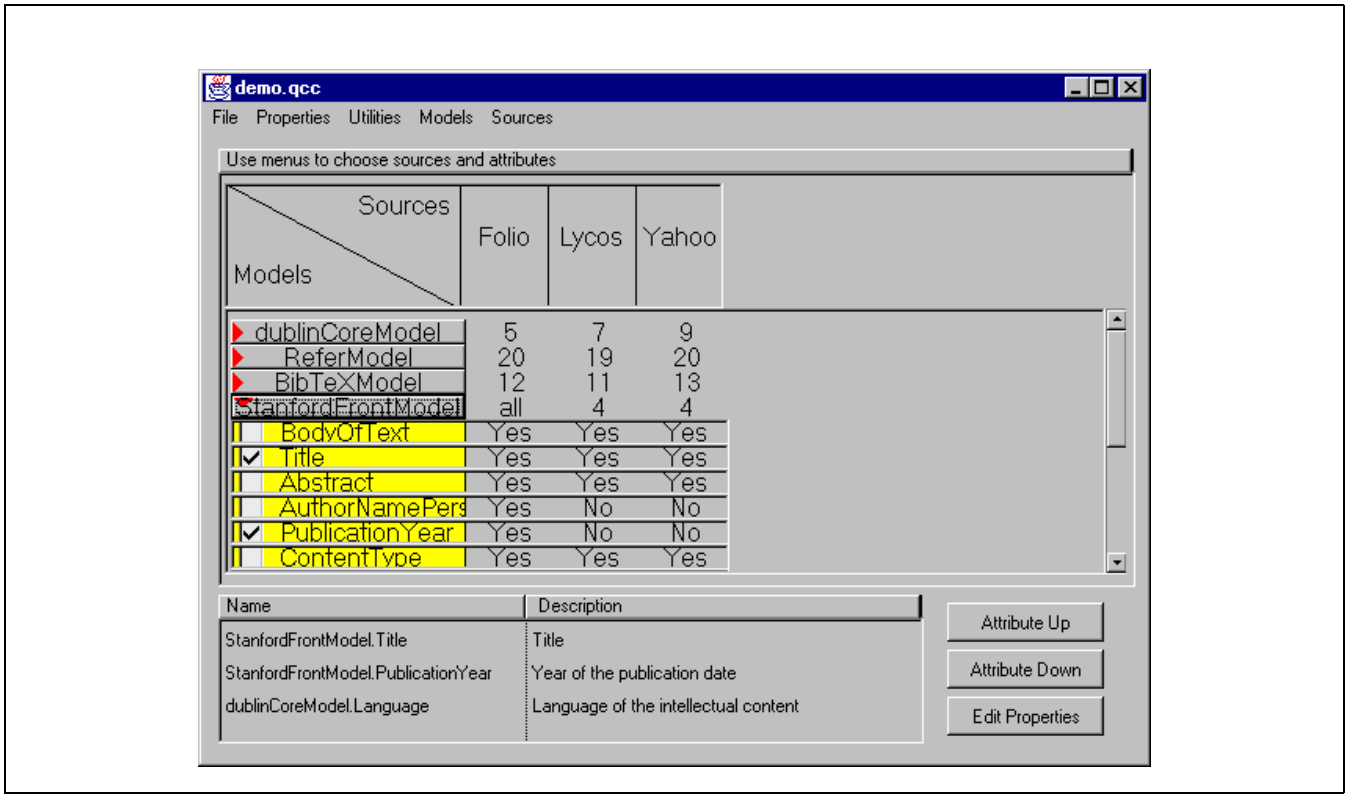

Figure 5: Browsing sources, attribute models, and attributes

constructor tool can additionally be enhanced if applets are allowed in place of input widgets.

Search operators are another complication in the interface design. The tradeoff is between exposure of the sources' full power on one hand, and simplicity for the most common search tasks on the other. Web search engines sometimes offer two interfaces, a simple, one-field form, and a more complex facility that provides more control over the search. The simple form usually involves no operators at all. Search terms are entered without the ability to limit keyword scope to fields. Usually, for the top-ranked result documents, all query terms are in effect implicitly connected through the **and** operator.

Our current constructor tool does not yet support designers in exploring the valid values and type for each selected field, although our metadata architecture does provide that information. However, the tool does allow designers to choose from a small palette of specialized input widgets to include in the synthesizer GUI. The default for any field selected by a designer is a simple text entry widget. However, plain text fields with no error correction may not be the best interface for many attributes. For instance, the designer might want to provide the patron with the convenience of a list box when filling in the "Language" field of a synthesizer, or might want to provide a map for helping the patron delineate a geographic region. To specify an input widget for an attribute (as well as to edit other attribute-specific information), the designer selects an attribute and clicks the "Edit Properties" button (bottom of Figure 5). The "Edit Field..." window shown in Figure 6 then pops up. It illustrates how a designer specifies that a specialized input widget be used for the selected attribute, and how the properties of the widget are edited. This example shows the editing of an input widget for specifying the language of documents to be retrieved. The designer can specify whether a synthesizer field is required or optional. In the example of Figure 6, the designer has specified that the language field is to use a list input widget. This choice has caused the List Box dialog to be exposed. It allows the designer to enter valid choices for the list widget. In this example, these are document languages that may be specified in queries.

The constructor tool supports a component software architecture for developers to add new input widgets. Like the list box widget of Figure 6, many custom input widgets will have associated property editors for setting information about the widgets. For instance, a range input type includes a minimum and maximum value to allow.

Our implementation is currently relatively simple in that this module uses a linear, pre-built layout scheme. The designer cannot currently arrange input widgets arbitrarily. Figure 7 shows the user interface of a very simple, finished synthesizer. It contains three fields, 'Title', 'Year', and 'Language'. When the patron runs the cursor over the 'Year'

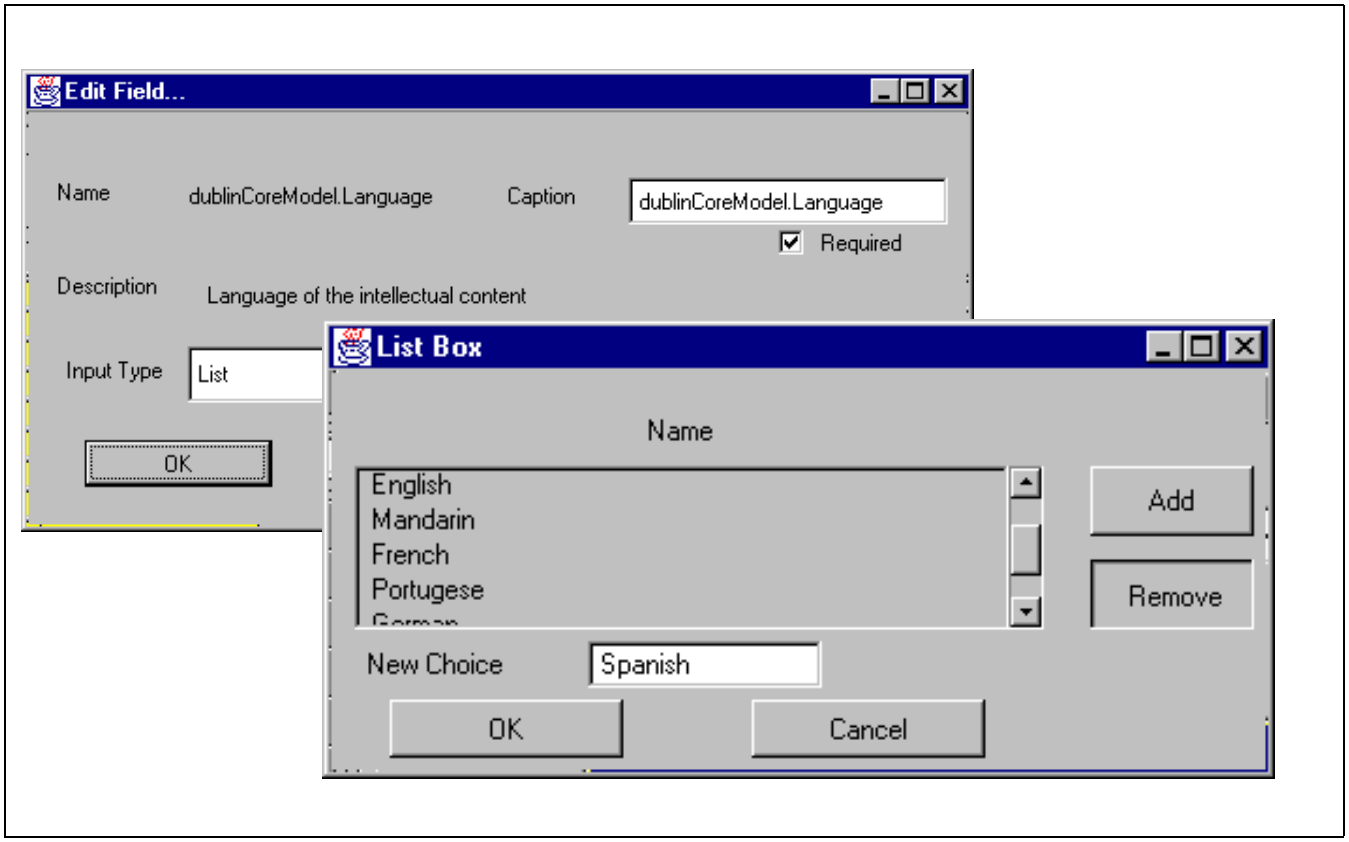

Figure 6: Choosing an input widget and interacting with its associated property editor

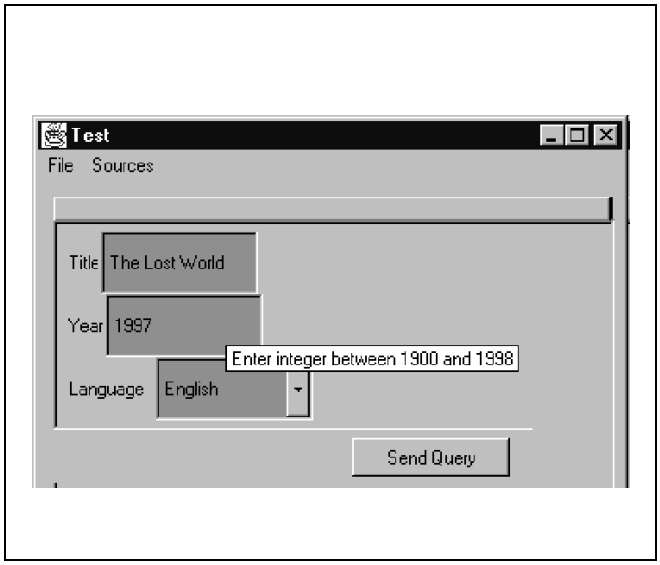

Figure 7: The patron interface of a very simple query synthesizer

field, balloon help informs the patron of the integer input range constraint associated with that field. Clicking on the 'Send Query' button constructs a query from the field values. Depending on the system in which the synthesizer is embedded, the query is then sent to a set of sources, or the patron may specify target sources herself at that point. In the case of our InfoBus, query and attribute translations take place automatically.

At this time, our constructor tool performs an implicit 'and' operation among the fields of a synthesizer to generate the final query. We plan to address the question of search operators and other transformations of the user inputs to a final query through the development of a simple scripting language (see the summary section for a sketch of our plans).

4.1 User interface discussion

The figures above give the reader a flavor of the current user interface for our constructor tool. Like others (e.g. [11, 5]), we are using structured forms as our patron interface. One of the contributions described in this paper is the tight integration of the form generation machinery with extensive metadata analysis and translation tools, and the focus on access to multiple, heterogeneous sources.

In this section, we explore two important aspects of the constructor tool's interface:

- how the user interface enables the designer to browse the relationships among sources, attribute models and attributes; and
- how the user interface encourages iterative design

Relationship browsing: The designer of the query synthesizer wants to understand relationships among attributes, attribute models, and sources. The first relationship is the ownership relation between attribute models and attributes. The Macintosh Finder, Windows Explorer, and various outlining applications all use a similar representation of this relationship—textual elements that can be selectively opened and closed. Accordingly, we adopted the same representation. The next two relations that are important to synthesizer designers are both support relationships. A designer would often like to know what attribute models and specific attributes are supported by specific sources, as well as the inverse. Our choice of a Finder-style representation for the ownership relation allowed us to model both of these relations in the same table. Columns in the table correspond to sources, while rows correspond both to attribute models and to individual attributes. The Finder-style widget makes the hierarchal relation among models and attributes explicit in the interface so there is no confusion over the semantics of a table cell. This approach allows the designer to determine the types of attribute models that are of interest before running through every attribute in detail. Furthermore, we expect that designers will interact with only a few attribute models at a time. The model representation allows the designer to reveal or hide attributes as convenient.

Screen space is at a premium for a two dimensional interface because designers want to see as much of the grid on screen at once as possible. This space concern led to our choice of menus to display and select the available attribute models and sources. Very rarely do all sources and models interest a designer at the same time. Selective display of information conserves space and eliminates cognitive clutter from the interface.

Iterative design: The importance of iterative design is widely recognized in design disciplines. To support a synthesizer designer in rapidly experimenting with different designs, the constructor tool includes a facility for interactively testing the query synthesizer under construction. This testing facility can be used at any time during construction. It also eliminates the need for an edit-compile-test cycle. In particular, our facility allows the designer to choose specific sources, enter sample values, and view the ensuing results. The result window separates results by source in a tree. Separating results from different sources makes it easier for the designer to analyze how well his query works on each source in practice.

5. TOOL ARCHITECTURE AND IMPLEMENTATION

Figure 3 shows that the constructor tool itself is subdivided into four modules, some of which communicate through the InfoBus with other facilities. The front-end of the constructor tool is implemented in Java.

The attribute-management module keeps track of attributes and the extent to which they are supported by the sources. The user-interface layout management module allows designers to build the forms interactively. The test engine communicates with the InfoBus to perform its work. The code generator will eventually produce both stand-alone Java input form interpreters, and forms integrated into our DLITE digital-library interface [4].

We use an object model with inheritance for input widgets, thus ensuring that the tool is extensible. In particular, the component model requires all input types to inherit from a common base class with methods for input validation, extraction, persistence, and error reporting.

Input validation involves checking specified constraints. For instance, a range input will check the given value to make sure that it lies between permissible limits. Input validation also checks to make sure that at least some input exists if the given attribute is required for a particular query.

Input extraction involves processing an input value for delivery to the query engine. For instance, a map click on “Belgium” might be converted to the string “Belgium” by the input extraction method associated with a map widget. By delegating responsibility to the input widget for mapping input values to query values, our model provides for data transformation. For instance, a developer could build an input widget that removes punctuation from text before delivering it to the query processing engine.

Input types maintain persistence by implementing the get and set methods. The resulting files are stored on the server rather than the client because of Java’s security model.

Input widgets are also required to implement a standard method for error reporting that returns an error string if the input entered by a user is not acceptable. Note that input types may not develop their own custom error reporting dialogs, so that interaction is consistent across different widget types.

Our approach to adding input widgets differs from that of existing library query synthesizers. Many systems hardwire input widgets specifically designed for a particular system to achieve sufficient integration between input widgets and the synthesizer, or they require the designer of the synthesizer to do some programming. Our approach allows query synthesizer designers to add prefabricated widgets without writing code.

Finally, we note that our constructor tool is extensible in three ways: it automatically integrates new attribute models as they become available, it finds and queries target sources for their schemas, and it can manually be extended to include new synthesizer field input widgets for specialized targets, such as geographical information systems.

6. STATUS AND FUTURE WORK

Our two-tier approach to query formulation allows the interactive design of targeted synthesizers that codify domain or task knowledge. In the first version of our prototype

system we have demonstrated some of those aspects. The system implements attribute models, the metadata repository, query syntax translation, and simple attribute translation facilities. These are used 'behind the scenes' by the constructor tool. All menus and table displays involving metadata are constructed at runtime, based on the information obtained through these metadata facilities. Our first version is still missing the code generator that creates final synthesizer output forms for integration with our DLITE digital library interface. The form shown in Figure 7 is a stand-alone facility.

Completion of version 1 will also require us to flesh out the metadata information provided by our search proxies. They currently do not provide all the proper information. We also need to add more attribute translation facilities than are currently part of the system. We do not expect these activities to be difficult.

The evaluation of version 1 will answer one particularly interesting question: How generic can we make our query synthesizers? Recall that synthesizer designers use metadata about expected target sources as guidelines when deciding which attributes to include in the synthesizer. Given our attribute translation facilities, we hope that the queries produced by the synthesizers will be applicable to sources other than the ones anticipated by the designer. While such genericity might not extend to sources of radically different content and organization, we hope that unanticipated sources more similar to the ones considered by the designer will indeed be usable through our synthesizers. Given the narrow task focus of our synthesizers, the use of 'similar' target sources with any given synthesizer is likely. Our experiments on the version 1 prototype will tell us just how far genericity can be extended through attribute translation.

For the second version of our prototype we plan a variety of extensions. First, we need to enhance our set of input widgets and, in doing so, test the extensibility of the widget pool. The current set is quite appropriate for generating text-input synthesizers. However, the constructor tool is ready to be taken beyond text. In particular, we would like the ability to use Java applets in place of standard input widgets. This will greatly enhance the constructor's ability to generate sophisticated and interesting synthesizers. For example, we would like the ability to create Java widgets that input values by displaying graphics, such as maps, and that generate properly translated values from the coordinates patrons point to. Such values might be the name of the closest city on a map, or the nearest face in a group picture.

A second enhancement concerns attribute translation. In version 1, translation occurs when a synthesizer emits a query after a patron has filled in the query form. The designer of the synthesizer is not informed of possible translations at the time she designs the synthesizer. We plan to allow the designer to invoke attribute translation as part of the synthesizer design phase. This will allow the designer to gauge more directly how widely any given attribute will be applicable to multiple target sources. For example, we might allow the designer to select any given attribute in the constructor interface. The tool would then highlight all the sources for which the attribute can be successfully translated.

In this first version, we have not included enough support for flexibility in using operators. Currently, our query synthesizers assume that all input fields are connected through **`and`**. Clearly, more sophisticated operators need to be made accessible to the patron. The underlying query translation machinery can manage a much richer set. Our current plan is to design a simple scripting language in which designers can specify how a query should be built from user input values. Most likely, we will allow designers to produce query expressions involving variable names that are later bound to values patrons enter into input fields. A crude example to explain the intent might be **`$FirstName NextTo $LastName and PY = $PubYear`**. Assume that later on, the patron specifies 'Richard', 'Nixon', and '1972' in the first/last name, and publication year fields respectively. The above script would be resolved to 'Richard(W)Nixon and py=1972'. Our existing query translation facility would in turn translate this query to native target query languages of other information sources. Remember that this query composition from fields will be specified by the *designer*, and will be exposed to the patron only to the extent determined by the designer through text placed on the input form. The actual composition will occur 'under the covers'.

A longer term enhancement will be to explore the construction of synthesizers that include query refinement. Most patrons do not produce a single 'killer query'. Instead, they start with one query and then refine it. Of course, the query synthesizers produced by our synthesizer constructor can be used for refinement, in that the contents of the input fields can be modified, but more sophisticated facilities should be made available.

7. REFERENCES

[1] Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke. Metadata for Digital Libraries: Architecture and Design Rationale. In *Proceedings of the Fourth Annual Conference on the Theory and Practice of Digital Libraries*, 1997. At http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1997-0055.

[2] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4), 1986.

[3] Chen-Chuan K. Chang and Hector Garcia-Molina. Evaluating the Cost of Boolean Query Mapping. In *Proceedings of the Fourth Annual Conference on the Theory and Practice of Digital Libraries*, 1997. At

http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1997-0053.

[4] Steve B. Cousins, Andreas Paepcke, Terry Winograd, Eric A. Bier, and Ken Pier. The Digital Library Integrated Task Environment (DLITE). In *Proceedings of the Fourth Annual Conference on the Theory and Practice of Digital Libraries*, 1997. Accessible at http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0049.

[5] D.W. Embley. NFQL: The Natural Forms Query Language. *ACM Transactions on Database Systems*, 14(2):168-211, June, 1989.

[6] *Information Retrieval: Application Service Definition and Protocol Specification*. ANSI/NISO, April, 1995. Preliminary Final Text.

[7] Andreas Paepcke, Steve B. Cousins, Héctor García-Molina, Scott W. Hassan, Steven K. Ketchpel, Martin Röscheisen, and Terry Winograd. Using Distributed Objects for Digital Library Interoperability. *IEEE Computer Magazine*, 29(5):61–68, May, 1996.

[8] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W.T. Lin, and Eugene Wong. Multibase – integrating heterogeneous distributed database systems. In *AFIPS National Computer Conf.* 1981.

[9] Jean Godby, Eric Miller Ron Daniel Stuart Weibel. OCLC/NCSA Metadata Workshop Report. March, 1995.

[10] USMARC Format for Bibliographic Data: Including Guidelines for Content Designation. Cataloging Distribution Service, Library of Congress, Washington, D.C., 1994.

[11] M.M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal*, 16(4):324-343, 1977.