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 produces 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. In this paper, we discuss how our tool for constructing synthesizers can facilitate the discovery of available attributes (e.g., 'title'), the collation of schemas from different sources, the selection of input widgets for a synthesizer (e.g., a drop-down list widget to support input of controlled vocabulary), and other design aspects. We also describe the user interface of our prototype constructor, which is 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

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. In this paper, we consider sources that accept vector-space queries as well as sources that accept traditional Boolean queries. A patron must understand the source well enough to know what operators to use in querying the source, whether or not it allows (or requires) the specification of fields, and if so, 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 must be of a given type (e.g., integers, not strings).

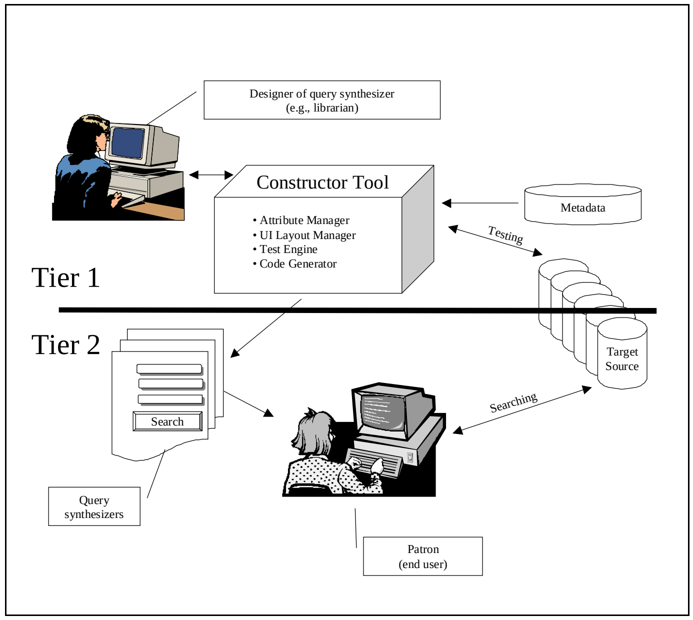

Figure 1. Two-tier approach to query formulation

In this paper, we propose a two-tier approach to facilitating query formulation in digital libraries (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 produces a graphical user interface (GUI) in which a digital-library patron (end user) can specify

* New contact information for Michelle Baldonado: Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304. E-mail: baldonado@parc.xerox.com

both a high-level, fielded query and a set of diverse target sources for that query (see Figure 2). 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 for two classes of designers to codify knowledge about tasks and sources ahead of time for the patron. The first class of designer is already familiar with many sources (e.g., a professional librarian) and thus uses the tool mostly to compare source features. The second class of designer is not already familiar with many sources and thus uses the tool not only to compare source features, but also to learn about sources. In either case, it is the patron who benefits from the designer's work because the patron can now more easily generate sophisticated queries without carefully investigating the sources.

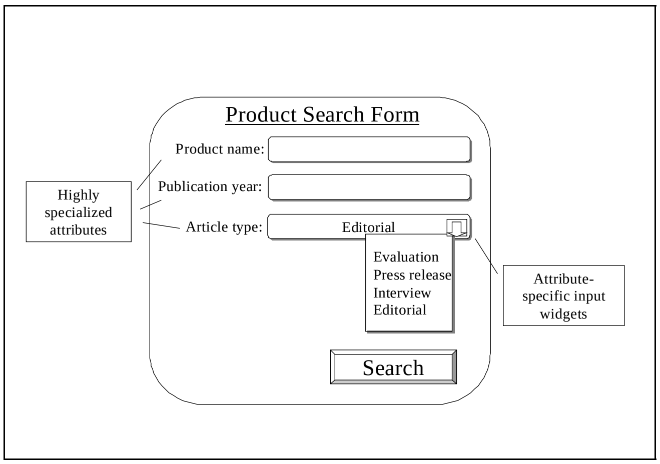

Figure 2. A simple GUI produced by a query synthesizer for product search

The context for our approach is the Stanford Digital Library project, which provides uniform access via the InfoBus [10] 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 3 presents an overview of the Stanford Digital Library architecture, including the constructor tool and its elements.

Very briefly, the constructor and synthesizers are used as follows. The designer first establishes the task for which patrons will use the new synthesizer. She 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 fields that the patron will be allowed to enter, specifies their formats, associates each field with a GUI element, and stipulates how the patron's inputs will be incorporated into the final query. The designer can also "test run" queries to ensure the design is acceptable. Finally, when the patron uses the resulting synthesizer, he simply interacts with the GUI produced, without having to understand the specifics of the chosen sources.

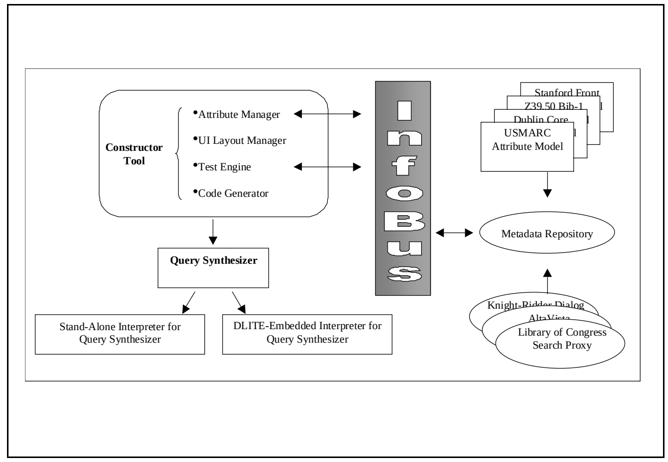

Figure 3. Stanford Digital Library Architecture, including the constructor tool

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 examine the schemas of all the relevant sources?
- 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?

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 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 "notebook computer." In contrast, the result set is limited to a meaningful couple of dozens when the query specifies that the results should have 'type' evaluation and 'publication year' 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. Fielded queries are not limited to sources that accept Boolean queries. Some sources that use statistical techniques to process queries (e.g., Verity databases) also allow for fielded searching. Furthermore, even 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's MARC scheme [14], Z39.50's BIB1 [9], or, more recently, Dublin Core [13]. 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, 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. Additional information is needed, such as what operators are relevant for an attribute or what data type is specified 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. Information about an attribute's data type can be used not only to guide the patron in formulating a query, but also to normalize the results that are returned. For example, a good system might present all dates in one uniform format. In order to accomplish this, the formats of information in each field must be accessible to the result preparation facilities. We will not discuss the details of such translations in this paper. Possible approaches are discussed in [11, 5].

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 constructor-tool prototype, we have addressed the problem of schema access by using our comprehensive metadata architecture that allows for the cataloging, 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. 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 contain the phrase 'author' in their documentation, or we can find out what data type is specified for a particular attribute. Attribute descriptions are particularly useful to synthesizer designers because they can highlight cases where attributes in different models have the same name, but have different meanings. For example, the 'population size' of a city may in one source include only the city center, in other cases the surrounding suburbs as well. The designer can make such differences clear to the patron by choosing descriptive labels for the respective fields.

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 of 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' and attribute models' 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.

Our constructor tool is extensible because it interacts with the Stanford metadata architecture. As new sources and attribute models are added to the InfoBus, they will be dynamically available to the constructor tool.

SCHEMA COLLATION

If a query synthesizer is destined to be used with a single source only, schema access facilities often suffice in helping the information expert choose which attributes to make available in the synthesizer.[1] 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. The designer must also specify constraints on the values patrons may enter for each field. The runtime system for the synthesizers must enforce these constraints once the synthesizer is deployed. Finally, the designer must specify a strategy for merging results from different sources

1. Note that some designers may opt to create a new schema even when only a single schema is supported by the chosen sources. See [16] for a discussion of transformations from USMARC to a logic-based ontology.

and ranking them for the patron. The derivation of a meaningful combined ranking is often difficult to accomplish because different sources use very different ranking criteria which are often kept as trade secrets. Approaches to rank merging are discussed in [8, 7, 15].

This section explores various possible approaches to schema collation: presenting the patron with a **union schema**, a **global schema**, an **intersection schema**, or a **regional schema**. Figure 4 illustrates each collation approach; Table 1 summarizes the distinctive features of each strategy. Note that the table suggests other schema collation strategies not considered here due to space restrictions. Several of these approaches have been developed by the database community over the years. See, for example, [3] for a survey. We have found that digital-library usage differs from database usage in that it tends to require a less rigorous, but more flexible approach to this problem.

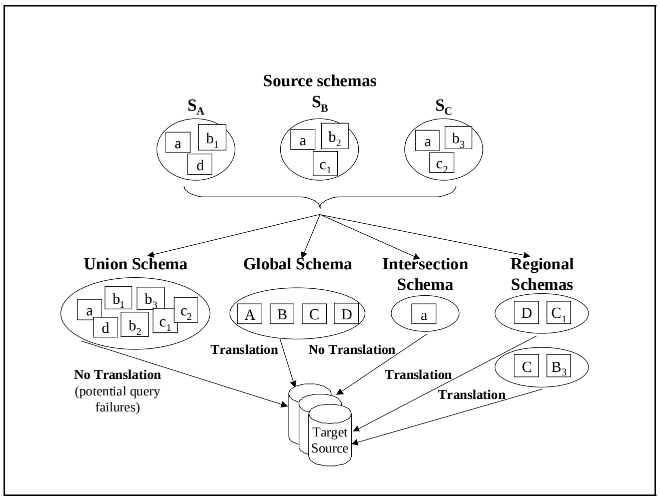

Figure 4. Different solutions for schema collation
Lower-case letters refer to source-specific attributes. Upper-case letters are newly defined attributes. Case and subscript variants of a letter (e.g., B, b_1) are similar and can be translated into each other.

Table 1: Comparison of schema collation strategies

	Runtime Collation	Runtime Translation	Maximally spanning set of attributes
Union	yes	no	yes
Global	no	yes	yes
Intersection	yes	no	no
Extended intersection	yes	yes	no
Regional	no	yes	no

Union schema: A simple approach to schema collation is to take the union of the selected sources' attributes at runtime and to present all of the resulting attributes to the patron. The advantage is that this is straightforward computationally (the synthesizer designer need not make any specific collation decisions ahead of time), 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 patron. Another problem is that incompatibilities can lead to query failures because not all attributes of the union are supported at all sources.

Global schema: In this approach, the synthesizer designer manually formulates a global schema for use by the end patron. At runtime, translation facilities are used to map queries expressed in the global schema to source-specific queries. An example of this approach can be found in [12]. Typically, the goal in creating a global schema is to provide maximal coverage of the attributes found in the sources under consideration. Thus, the designer of a global schema often takes into account the semantic equivalence of attributes. In Figure 4, the global schema created for source schemas S_A, S_B, and S_C includes two attributes that correspond to an "equivalence" class of attributes: namely, ***B*** corresponds to $\boldsymbol{b_1}$, $\boldsymbol{b_2}$, and $\boldsymbol{b_3}$, while ***C*** corresponds to $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$. Note again that even though two attributes may have the same name, they may nevertheless need to be treated as different attributes because their values have different meanings.

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 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 number', then the global schema needs to introduce a second 'product number' attribute in order to maintain a maximally spanning set of attributes. 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.

Intersection schema: A third approach is to use the intersection of target schemas: at runtime, only those attributes that are supported by all target sources of interest are presented to the patron. The advantages include both ease of computation and reduction of the number of attributes presented to the patron. 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.

The intersection approach can be extended 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 schema (***a***) might be enriched by attribute $\boldsymbol{b_1}$, which is then translated to $\boldsymbol{b_2}$ and $\boldsymbol{b_3}$ 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 represented by a single attribute if their semantic equivalence can be recognized. 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.

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 (e.g., SenseMaker [2]). We have discussed the relevant tradeoffs and limitations of this particular transformation technique elsewhere [4].

In the general case, discovery of semantic equivalence of attributes is very difficult to automate and is tedious to accomplish manually. In practice, this approach can be used successfully on high priority attribute models and attributes.

Regional schema: The final approach detailed in this section is the one that we have adopted. A regional schema, like a global schema, is formulated by a synthesizer designer. The goal of the designer in creating a regional schema is to develop a schema that is useful for a particular task or domain, rather than to develop a schema that maximally spans the attributes supported by the target sources. For example, a global schema that is developed for several book databases will include ISBN number if that attribute (or a variant of it) is available at all of the selected sources. In contrast, a regional schema that is developed for the same sources might forego that attribute if it is not deemed useful for the expected task (perhaps patrons will only perform queries when they are looking up bibliography references and will never have ISBN numbers).

In fact, regional schemas need not reliably cover all possible target sources. The "region" of a regional schema is the set of schemas from which it is derived. In Figure 4, the region for the top schema is S_A and S_B, while the region for the bottom schema is 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. Patrons who use the resulting synthesizers and submit the resulting queries to unanticipated sources may find that this strategy works well, thanks to the translation techniques described in the section on intersection schemas. At other times, the strategy may have failings. 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.

Our own experience with global schemas has been the motivation for us to switch to regional schemas. Before we designed and implemented the constructor tool described in this paper, all of our query synthesizers used USMARC as a 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

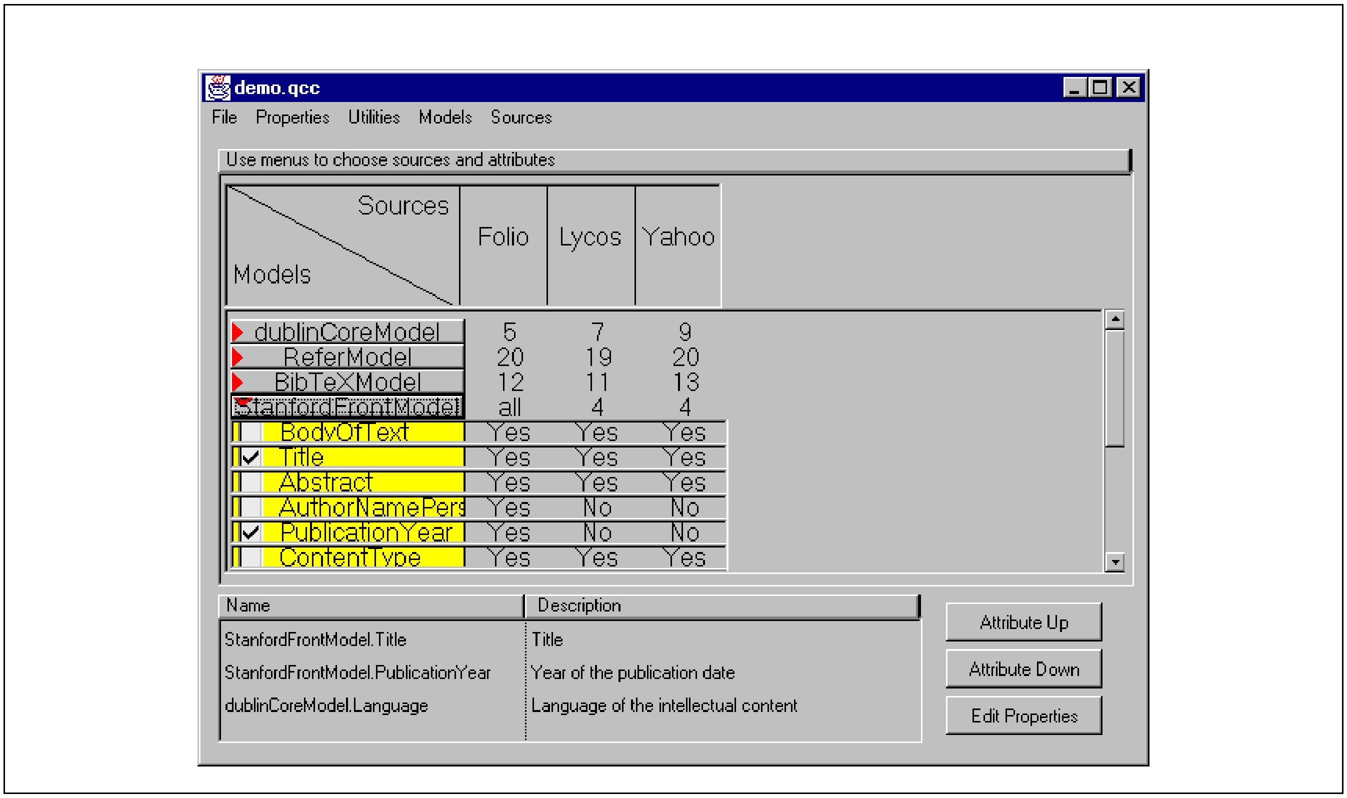

Figure 5. Browsing sources, attribute models, and attributes

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.

In deciding to allow for regional schemas, the need for a constructor tool became apparent. The constructor tool described here allows an information expert to perform the metadata browsing necessary to create a synthesizer containing task-specific attributes that are useful for the target sources. As will be shown later, the constructor tool 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 both regional schemas and synthesizer interfaces.

CONSTRUCTOR TOOL USER INTERFACE

This section surveys the user-interface design requirements we set for the constructor tool and explains how our current version of the constructor tool addresses these requirements

Requirement 1: Allow the designer to explore the relationships between sources and attribute models in order to construct a useful regional schema.

Our approach to requirement 1: Figure 5 is a screen shot taken of the current version of the constructor tool. When the tool is first launched, the synthesizer designer selects sources and attribute models from their 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). Each cell in the table describes how many

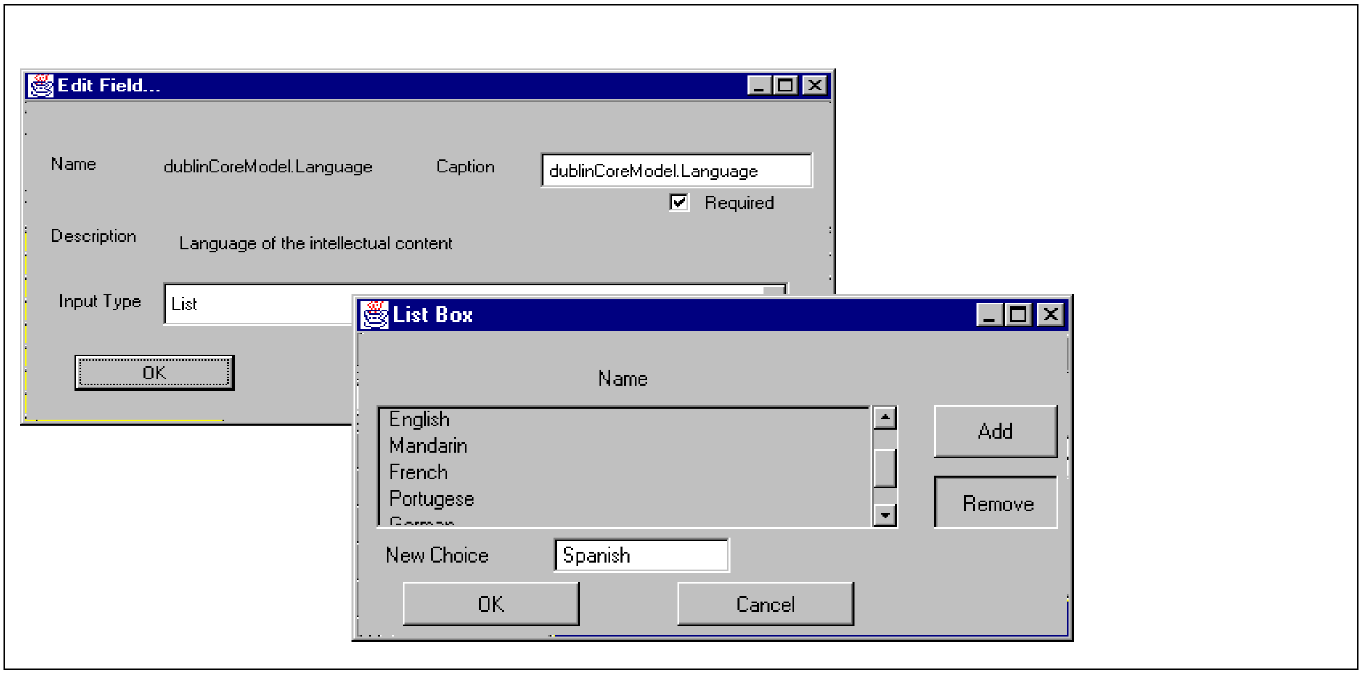

Figure 6. Choosing an input widget (above) and interacting with its associated property editor (below)

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. In Figure 5, an attribute model has been "opened" to reveal its member attributes: The entries below 'StanfordFrontModel' show attributes such as 'Title' and 'Abstract'. 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.

Note that the Macintosh Finder, Windows Explorer, and various outlining applications all use a similar technique for showing hierarchical relationships. Our use of a Finder-style widget for attribute models makes it possible to see three relationships (the relationships between attribute model and attributes, between attribute models and sources, and between attributes and sources) in a single table. Furthermore, this approach allows the designer to determine the types of attribute models that are of interest before running through every attribute in detail. As 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. Selective display of information conserves space and eliminates cognitive clutter from the interface.

Requirement 2: Allow the designer to specify a GUI element for each attribute in the newly developed regional schema. While text-based synthesizers (illustrated by Figure 2) are useful for many domains, more complex synthesizers might include GUI elements such as pull-down menus, images, and maps. 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.

Our approach to requirement 2: Selecting an attribute from the constructor tool's table (by checking its associated check box) causes the attribute to appear in the lower panel (see Figure 5). At this point, the designer chooses 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. To specify a specialized input widget for an attribute (as well as to edit other attribute-specific information, such as whether or not the field will be required for the patron), the designer selects an attribute and clicks the "Edit Properties" button (bottom of Figure 5). The "Edit Field..." window shown in Figure 6 illustrates a designer's decision to use a list input widget for a 'Language' field. This choice has caused the List Box property editor to appear (also show in Figure 6). In this case, the property editor asks the designer to enter valid choices for the list widget. In this example, these are the document languages that are appropriate for the patrons' expected tasks.

Our 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.

Requirement 3: Allow the designer to specify value constraints (where applicable).

Our approach to requirement 3: As we hinted in the discussion of the previous requirement, our constructor tool allows the designer to specify value constraints for an attribute in the property editor for its associated input widget. The widget then enforces its constraints and informs the patron of those constraints. For example, when the patron runs the cursor over the 'Year' field in Figure 7, balloon help informs the patron of the integer input range constraint associated with that field.

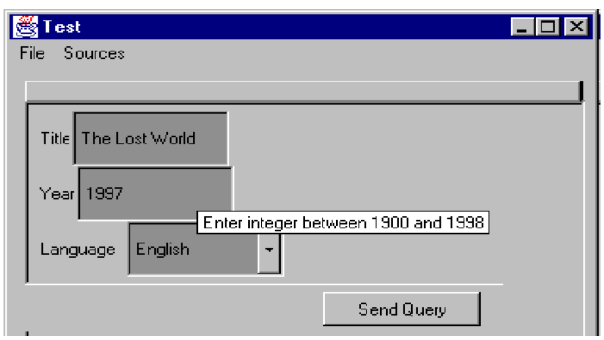

Figure 7. The GUI of a simple query synthesizer

Requirement 4: Allow the designer to specify the layout of the chosen GUI elements.

Our approach to requirement 4: The order in which the attributes appear in the lower panel of Figure 5 corresponds to the order in which they will appear in the generated query synthesizer. Accordingly, the tool provides buttons that can be used to edit this order

Our implementation is currently relatively simple in that we use 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'.

Requirement 5: Allow the designer to dictate how the user's interaction with the GUI should shape the query under construction. For example, consider designing a query synthesizer to be used for accessing statistics about California. Some of the sources might in fact be national-level statistical databases that expect queries about cities to include both city information and state information. In the context of this query synthesizer, the designer might choose to let patrons enter cities only and then to preprocess those city values to append the information that these are California cities.

Deciding where to insert search operators in the constructed query is also an important issue for the designer. The trade-off 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 approach to requirement 5: 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 patrons' inputs to a final query through the development of a simple scripting language (see the summary section for a sketch of our plans).

Requirement 6: Allow the designer to engage in iterative design.

Our approach to requirement 6: 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 initiate testing from the menu shown in Figure 5, then choose specific sources, enter sample values, and view the ensuing results. The result window organizes results by source. The designer can focus her attention on one source at a time to investigate how the front-end query was received by that source.

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 end GUI 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 [6].

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.

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.

The evaluation of version 1 will answer one particularly interesting question: how generic are 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 and the notion of regional schemas, the queries produced by the synthesizers will be applicable to sources other than the ones anticipated by the designer. While it is unlikely that a synthesizer designed with one set of sources in mind will extend to sources of radically different content and organization, we hope that unanticipated, but similar, sources will indeed be accommodated easily

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 (or Java Beans) 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 with ‘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 NextTo 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 during the processing of the patron's input.

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 can be made available.

Although many challenges remain in furthering the functionality of our constructor tool, we are encouraged in this experiment in using 'live' metadata access to support the semi-automatic construction of query input facilities. We are committed to building and extending tools that support intermediaries and simplify the tasks of patrons.

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 Second ACM International Conference on Digital Libraries*, 1997.

2. Michelle Q Wang Baldonado and Terry Winograd. SenseMaker: An Information-Exploration Interface Supporting the Contextual Evolution of a User's Interests. In *Proceedings of the Conference on Human Factors in Computing Systems*, 1997

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

4. Chen-Chuan K. Chang and Hector Garcia-Molina. Evaluating the Cost of Boolean Query Mapping. In *Proceedings of the Second ACM International Conference on Digital Libraries*, 1997.

5. Chen-Chuan K. Chang and Héctor García-Molina. Conjunctive Constraint Mapping for Data Translation. In *Proceedings of the Third ACM International Conference on Digital Libraries*, 1998.

6. Steve B. Cousins, Andreas Paepcke, Terry Winograd, Eric A. Bier, and Ken Pier. The Digital Library Integrated Task Environment (DLITE). In *Proceedings of the Second ACM International Conference on Digital Libraries*, 1997.

7. Ronald Fagin. Combining Fuzzy Information from Multiple Systems. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1996.

8. Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching. In *Proceedings of the International Conference on Management of Data*, 1997.

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

10. 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.

11. Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *Transactions on Database Systems*, 19(2):254–290, June, 1994.

12. 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.

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

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

15. Ellen M. Voorhees and Richard M. Tong. Multiple Search Engines in Database Merging. In *Proceedings of the Second ACM International Conference on Digital Libraries*, 1997.

16. Peter C. Weinstein. Ontology-Based Metadata: Transforming the MARC Legacy. In *Proceedings of the Third ACM International Conference on Digital Libraries*, 1998.