International Journal of Digital Libraries Manuscript Nr.
(will be inserted by hand later)

The Stanford Digital Library Metadata Architecture [*]

Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke

Computer Science Department
Stanford University
Stanford, CA 94305-9040, USA
{michelle,kevin,gravano,paepcke}@db.stanford.edu
Phone: +1-415-723-9684
FAX: +1-415-725-2588

Abstract. The overall goal of the Stanford Digital Library project is to provide an infrastructure that affords interoperability among heterogeneous, autonomous digital library services. These services include both search services and remotely usable information processing facilities. In this paper, we survey and categorize the metadata required for a diverse set of Stanford Digital Library services that we have built. We then propose an extensible metadata architecture that meets these requirements.

Our metadata architecture fits into our established infrastructure and promotes interoperability among existing and de-facto metadata standards. Several pieces of this architecture are implemented; others are under construction. The architecture includes attribute model proxies, attribute model translation services, metadata information facilities for search services, and local metadata repositories. In presenting and discussing the pieces of the architecture, we show how they address our motivating requirements. Together, these components provide, exchange, and describe metadata for information objects and metadata for information services. We also consider how our architecture relates to prior, relevant work on these two types of metadata.

Keywords: metadata architecture, interoperability, attribute model, attribute model translation, metadata repository, InfoBus, proxy architecture, heterogeneity, digital libraries, metadata survey

[*] This material is based upon work supported by the National Science Foundation under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by DARPA, NASA, and the industrial partners of the Stanford Digital Libraries Project. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the other sponsors.

1 Introduction

As traditional libraries have evolved to meet the needs of their patrons, librarians have encountered and addressed a host of metadata-related issues. Today, sophisticated library cataloging principles and schemes help all of us in finding the information that we need in our local library. As work on digital libraries progresses, however, new metadata needs are arising. The increased ease with which a user can cross the boundaries from one "library" to another, the ability to organize online contents into complex structures, and the development of tools that let users transform those structures, all call for a rethinking of what metadata is and how it can be shared in digital libraries.

In the Stanford Digital Library project, we view long-term digital library systems as collections of widely distributed, autonomously maintained services. Of course, a digital library system must include services that allow users to search over collections of information objects. Examples of searchable collections include traditional library collections, digital images, e-mail archives, video, on-line books, and scientific article citation catalogs (containing only metadata about the articles, not the articles themselves). While searching services are valuable, they are not the only kind of service in the digital library of the future. Remotely usable information processing facilities are also important digital library services. These services provide support for activities such as document summarization, indexing, collaborative annotation, format conversion, bibliography maintenance, and copyright clearance.

Our project has focused on developing an infrastructure in which these disparate services can communicate and interoperate with one another. Specifically, the goal of our digital library testbed is to provide an infrastructure that affords interoperability among these heterogeneous, autonomous components, much like a hardware bus enables interaction between disparate hardware elements. Our assumption is that inventing a new standard is unlikely to solve the interoperability problem. Instead, we propose that InfoBus components use proxies to access and communicate with each other. Proxies (also called "wrappers") allow heterogeneous services to give the illusion that they respond to a standard set of methods. We call our proxy-based infrastructure the *InfoBus* [1].

This paper provides a framework for understanding the classes of metadata and range of metadata needs that are necessary for our InfoBus services. We outline and ground this framework in Section 2 by surveying our InfoBus services and analyzing the categories of metadata that they require. In particular, we give concrete examples of four very different InfoBus services. First, we show that our automated resource-discovery service needs metadata about what collections exist and what each collection contains. Second, we observe that our service for formulating queries appropriate for multiple sources relies upon metadata about how information objects are described within each source. Third, we explain why our service for translating queries requires protocol-related metadata about the selected search services. Finally, we analyze how our service for making sense of query results can utilize metadata about information objects and their underlying representations in order to compare them and to portray their surrounding context.

Our decision to design and implement a metadata architecture has grown out of this understanding of metadata needs. We have found that ad-hoc approaches to these metadata issues do not scale and cause problems for interoperability. Related work on metadata issues (discussed in Section 4) is relevant for specific metadata issues that we have encountered, but does not address the problem of integrating and sharing different types of metadata information in the ways that we require. Hence, we have designed our own metadata architecture. It rests on top of our established proxy-based framework and can thus be situated amidst existing and de-facto standards for metadata. Several pieces of this metadata architecture are implemented; others are under construction. We present the architecture in detail in Section 3 and show how its features map onto our concrete metadata requirements. Readers interested in the design rationale for this architecture should refer to a separate paper on this topic [2].

2 Our Metadata Requirements

In this section, we present the InfoBus services that have motivated and shaped our metadata architecture. The discussion of each service illustrates a number of concrete requirements for the architecture.

The services described in this section are interconnected in that they all address the problem of interacting with heterogeneous searchable collections. This problem is multi-faceted because it can involve a range of activities, including locating and selecting among relevant collections, retrieving information from these collections, interpreting the information retrieved from them, managing and organizing the retrieved information at a local level, and sharing this information with others [3, 4].

A simplified example scenario gives an overview of how metadata-related issues emerge. Imagine a user, Pat, who is interested in the topic of data mining. Pat's first task is to determine what searchable collections are relevant to this topic. Pat turns for help to a resource-discovery service (Section 2.1) that relies upon meta-information about each collection to answer the question. With a set of heterogeneous collections selected, Pat is ready to begin searching. The user interface service that Pat is using allows for queries to be entered in a rich Boolean front-end language. The query constructor (Section 2.2) provided by the user interface uses knowledge of the attributes supported by each search service in order to present Pat with a set of attributes that could be of value in the query. Once Pat has entered a query, it is up to a query translation service (Section 2.3) to translate this front-end query into its native equivalent for each selected collection. The query translator requires meta-information about the capabilities of each collection to accomplish this task. After the native queries have been issued, Pat receives back a set of results from the selected, heterogeneous collections. The result analysis component of the user interface then unifies these heterogeneous results and, under user guidance, constructs integrated overviews of them (Section 2.4). Pat can now make sense of these initial results and decide what to do next.

2.1 Automated Resource Discovery

End users have a large variety of searchable collections available to them. Accessing all these collections for each query is not practical. First, there may be too many such collections, with varying response times. Second, some collections charge for their access. Furthermore, presumably only a few collections contain useful items for a given query. Therefore, a crucial component of a Digital Library is a tool that assists users in discovering the useful resources for their queries.

Finding the best collections for a query is the goal of *GlOSS* [5, 6], a resource-discovery service within our Digital Library testbed. A user submits a query to *GlOSS*, and *GlOSS* returns a rank of the available collections. This rank is based on *estimates* of the expected number of hits for the query at each collection (Figure 1). Then, the user submits the query to the top collections, as determined by *GlOSS*. This way, the user avoids accessing the majority of the available collections, focusing the search on the most promising ones. *GlOSS* uses content summaries of all target collections to compute its estimated ranks, which brings up the problem of scalability. *GlOSS* needs to be able to handle content summaries of a large number of searchable collections. To keep these summaries small, *GlOSS* only stores *partial* information.

The *GlOSS* content summary of a collection consists of the words that appear in the collection, together with the number of items where each word occurs, as the following example illustrates. An unusual characteristic of the *GlOSS* metadata is that it is aggregate information

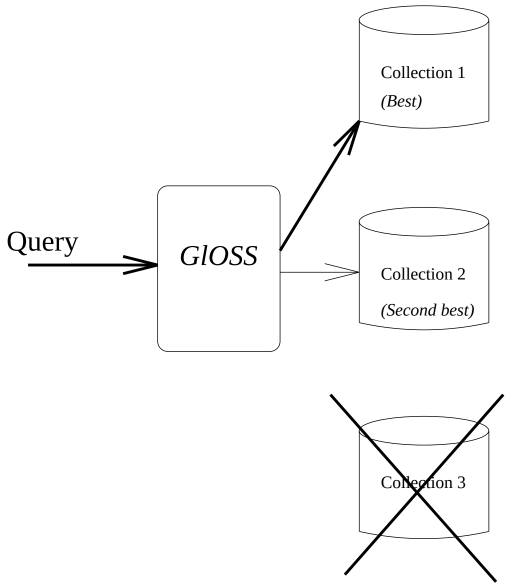


Fig. 1. Users submit queries to *GlOSS*, and *GlOSS* ranks the available collections accordingly.

about the whole collections, rather than being information about the individual items in the collections.

Example 1. Consider the following user query q:

`Title Contains mining`

Suppose that *GlOSS* has three collections available, c_1, c_2, and c_3. To rank these collections for q, *GlOSS* knows how many items match q at each collection. For example, *GlOSS* knows that c_1 has a total of 100 items that contain the word "mining" in their title, c_2 has 10 such items, and c_3 none. *GlOSS* will suggest c_1 as the best source for the query (100 matching items), and c_2 as the second best source (10 matching items). Source c_3 will not appear in the *GlOSS* rank.

GlOSS does not keep information on how words cooccur in the items. Hence, it needs to make assumptions on the distribution of query words in the collections whenever a query contains more than one word. These assumptions may not hold in reality, leading to wrong result-size estimates. However, the *GlOSS* information tends to be orders of magnitude smaller than the corresponding collections, facilitating scalability. Furthermore, experimental results with real-user queries showed that *GlOSS* ranks searchable collections correctly most of the time [5].

To suggest collections for a query, *GlOSS* extracts content summaries from each collection. These content summaries, like in the example above, include the vocabulary of each collection, together with frequency counts that are associated with them. The current implementation of *GlOSS* (available at `http://gloss.stanford.edu/`) gets the content summaries in ad hoc ways. For example, *GlOSS* extracts all the bibliographic entries from the CS-TR databases and processes these entries locally to produce the right content summaries. (CS-TR is an emerging digital library of computer science technical reports. See `http://www.ncstrl.org/`.) To get the summary of a collection of HTML documents, *GlOSS* could fetch all the documents in the collection by following links, and then extract the necessary information locally. However, this approach is expensive in terms of the number of accesses to the collection and the amount of information retrieved. Also, it would not work for collections of non-HTML documents that do not export their entire contents. Therefore, *GlOSS* needs the collections to collaborate and export their content summaries upon request. In summary, *GlOSS* requires the following of our metadata architecture:

- *We need a way to learn about the available collections.*
- *We need a way to extract the content summary associated with each collection.*

2.2 Query Formulation

Different searchable collections typically export different attributes for searching. In query formulation, an important role of the user interface is thus to make the user aware of what attributes are available for searching.

DLITE is a user interface service that has been developed for our Digital Library testbed [7, 8, 9]. It allows librarians or end users to specify *workcenters* for the various information-related tasks in which they frequently engage. A DLITE workcenter gathers together components and tools that a user might need to complete a task. The current DLITE workcenter for searching includes a query constructor that relies upon a minimal, canonical attribute model. This attribute model is unstructured and contains just the canonical attributes `Author`, `Title`, and `Subject`. By "unstructured," we mean that the attribute model is a flat name space. We will introduce more complex attribute models in Section 2.4. Figure 2 shows how the DLITE query constructor appears to the user. The user fills in as many of the fields as are desired, then hits the "Create Query" button. An object representing the query then appears in the circular area below the constructor. The user can take this query object and drop it on any of the represented search services (Dialog_275 and AltaVista in this example) for evaluation.

We can allow users to formulate queries that refer to these canonical attributes because we have tables that map canonical attributes (often approximately) onto their native collection attribute equivalents. However, this current approach does not scale well since it requires one table entry for each possible equivalence mapping. A more general approach would introduce independent attribute translators. With independent attribute translators in place, the query translation service could first find out what attributes are supported natively by a search service. If the necessary attributes are not supported, then either the query translation service or the search service itself could try to locate an attribute translator to do the mapping (attribute translators are likely to be built for a wide variety of mappings, but are not guaranteed to exist for all possible mappings).

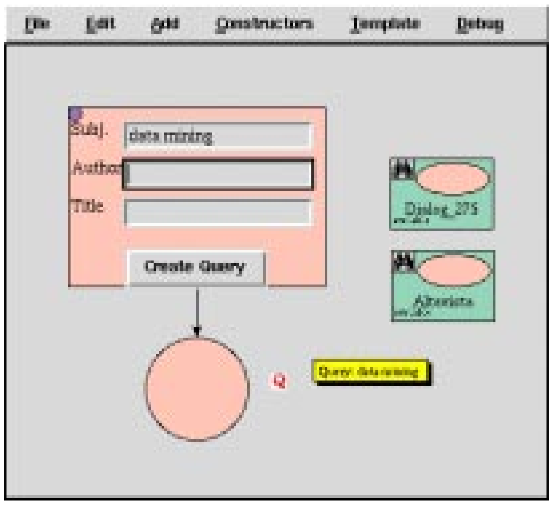


Fig. 2. The DLITE query constructor is shown here using a static set of canonical attributes. With a richer metadata model, it could be extended to use a dynamically determined set.

Additionally, the translation process could be decomposed into several intermediate translation steps, each of which could be carried out by a separate translator. Either the query translation service or an individual translator could take the initiative to compute this translation decomposition. This strategy reduces the number of equivalence mappings that must be recorded.

These query formulation needs correspond directly to requirements on our metadata architecture.

- *We need to include attribute models as first-class objects so that we can communicate with them and perform necessary transformations.*
- *We need facilities for finding out efficiently what attribute models and individual attributes are natively supported by each search service.*
- *We need attribute model translators that are capable of translating from canonical attributes and canonical attribute values into native attributes and native attribute values.*

By designing a metadata architecture that takes into account these requirements, we also have the building blocks for designing more sophisticated query constructors. It is unlikely that any one canonical attribute model would be sufficient for all levels of users, tasks, and collections. A metadata architecture that treats attribute models as first-class objects can easily accommodate a pool of several canonical attribute models. A DLITE search workcenter could use meta-information about each of these canonical attribute models to determine when a particular model is appropriate. Furthermore, it could tailor the user's view of the chosen canonical model by using information about the correspondences between each search service's native attributes and the canonical model's attributes. For example, if we think of the native attributes for each search service as corresponding to a set of canonical attributes, then we can imagine a query constructor that presents the user with the intersection of these canonical attribute sets. The automated query translation module could then use the attribute model translation services to prepare the user queries for delivery to the target collections.

2.3 Automated Query Translation

There are a wealth of search engines behind the collections in digital libraries, each with a different document model and query language. There have been various approaches to address this problem. The most typical way is to provide a front-end that supports the least common denominator of the underlying services to hide the heterogeneity [10, 11]. In contrast, our approach is to allow a user to compose Boolean queries in one rich front-end language [12, 13]. For each user query and target source, we transform the user query into a *subsuming query* that can be supported by the source but that may return extra documents. The results are then processed by a *filter query* to yield the correct final result. In summary, given a user query, the query translator generates a native query that returns a minimal number of extra documents and also a filter query to apply in post-processing. The following example illustrates our approach.

Example 2. Suppose that a user is interested in documents discussing data mining and data warehousing. Say the user's query is originally formulated as follows:

```
Title Contains warehousing AND data (W) mining
```

This query selects documents with the three given words in the `Title` field; furthermore, the `W` proximity operator specifies that the word "data" must immediately precede the word "mining."

Now assume the user wishes to query the INSPEC database offered by the Stanford University Folio system. First, the translator must map the front-end `Title` field onto the corresponding INSPEC field (as described in Section 2.2). Next, it must map the query operators. Unfortunately, this source does not understand the `W` operator. In this case, our approach will be to approximate the predicate "`data (W) mining`" by the closest predicate supported by Folio, "`data AND mining`." This predicate requires that the two words appear in matching documents, but in any position. Thus, the native query that is sent to Folio-INSPEC is:

```
Find Title warehousing AND data AND mining
```

Notice that now this query is expressed in the syntax understood by Folio. The native query will return a preliminary result set that is a super-set of what the user expects. Therefore, an additional post-filtering step is required at the front-end to eliminate all documents that do not have the words "data" and "mining" occurring next to each other. For this example, the required filter query is:

```
Title Contains data (W) mining
```

As illustrated in the above example, the major component of query translation is *query capability mapping* in which user queries are rewritten to be expressible with respect to the target services' capabilities. The query capability mapping process is therefore driven by the metadata that describe the query capabilities of the search services. In Example 2, the query translator needs metadata related to service functionality to answer the following questions: Is `Title` a searchable attribute? Can one search `Title` with the `Contains` operator? A service may only support the `Equals` operator for some attributes. Is the proximity operator `W` supported? If not, are there any semantically related operators to substitute? For instance, DEC's AltaVista supports the operator `NEAR`, which restricts search terms to be no more than 10 words apart, and is therefore a better substitute than `AND`. Moreover, the query translator also needs to check if the query words (i.e., `warehousing`, `data`, and `mining`) are stopwords: common words that are not indexed in the target system. If this is the case, the query may get no hits at all and the user may be advised of this fact.

In general, our algorithms for query capability mapping require metadata about the underlying search engines to perform a complete translation. Currently, we encode the required metadata (capability and schema definition) within the query translator. Consequently, the current implementation may not scale well. In addition, the query translator's metadata knowledge may not be up-to-date if the underlying services change. Therefore, to address these problems, it would be ideal to have services maintain and provide their own metadata in our architecture. In summary, the query translator needs the following metadata from search services:

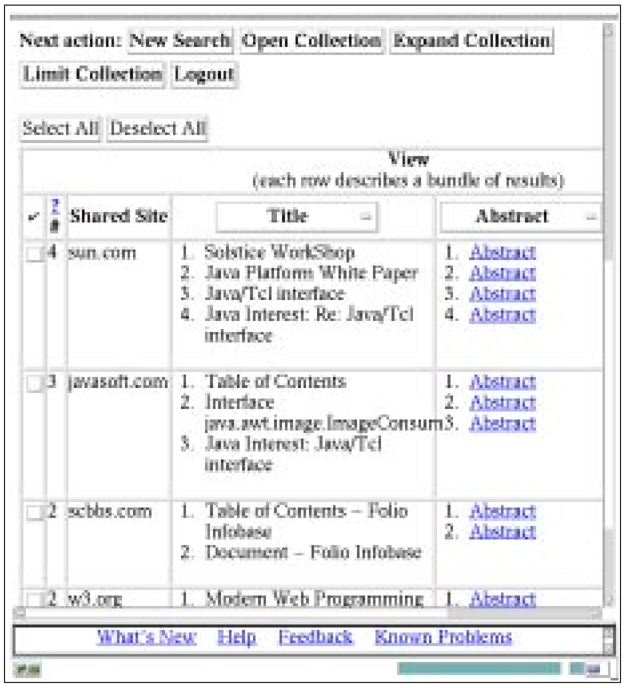
- *We need the schema definition: the set of searchable and/or retrievable attributes and the supported search methods (i.e., legal combinations of operators) for searchable attributes.*
- *We need to know the supported operators in addition to Boolean operators (e.g., the type of proximity operators supported).*
- *We need to be aware of the stopwords used.*
- *We need to know the vocabulary of the collection, i.e., the set of words indexed by the service. This is used, for instance, to enumerate words that match the stem of a particular word when stemming [14, 15] must be emulated because it is not a supported capability.*
- *We need to know the details of other features, e.g., for truncation, the supported truncation patterns.*

2.4 Result Analysis

The introduction of canonical attribute models into the metadata architecture is valuable not only for query formulation, but also for result analysis. Viewing heterogeneous results through the lens of a canonical attribute model allows users to obtain a unified overview of those results.

Making result analysis easier for the user is one goal of SenseMaker, another user interface service developed for our Digital Library testbed [9, 16]. SenseMaker allows users to experiment iteratively with different views of their results. Within a view, complexity is reduced in two ways. Similar results may be bundled together, and identical results may be merged together. Since many bundling and identity criteria are possible, users can try out different combinations in order to gain multiple perspectives on the set of results. Users also determine what attribute values should be displayed in each view.

Figure 3 shows a partial tabular view of results obtained by sending the keyword query `java development` to two technical article citation collections and two World-Wide Web collections. In this view, the displayed canonical attributes are `Shared Site`, `Title`, and `Abstract`. The bundling criterion specifies that results with `URL` values referring to the same Internet site should be bundled together (results with undefined `URL` values are put into their own bundle). Other possible bundling criteria include bundling together results with the same `Author` value, bundling together results with the same or similar `Title` values, and bundling together results from the same collection, among other criteria. For the view portrayed here, the identity criterion specifies that results with identical `Title` values should be treated as duplicates. Other possible identity criteria include merging results with identical `URL` values, merging results with identical `URL` values and identical `Title` values, and so on.


Next action: New Search Open Collection Expand Collection
Limit Collection Logout
Select All Deselect All
What's New Help Feedback Known Problems

View
(each row describes a bundle of results)

✓	? #	Shared Site	Title	Abstract
☐	4	sun.com	1. Solstice WorkShop 2. Java Platform White Paper 3. Java/Tcl interface 4. Java Interest: Re: Java/Tcl interface	1. Abstract 2. Abstract 3. Abstract 4. Abstract
☐	3	javasoft.com	1. Table of Contents 2. Interface java.awt.image.ImageConsum 3. Java Interest: Java/Tcl interface	1. Abstract 2. Abstract 3. Abstract
☐	2	scbbs.com	1. Table of Contents – Folio Infobase 2. Document – Folio Infobase	1. Abstract 2. Abstract
☐	2	w3.org	1. Modern Web Programming	1. Abstract

Fig. 3. SenseMaker Unified Result Overview

In SenseMaker, the user's choices of display attributes, bundling criterion, and identity criterion all depend upon the characteristics of the chosen collections. If a user chooses a World-Wide Web collection (e.g., AltaVista), then the list of possible display attributes would include `URL`, the list of bundling criteria would include bundling by Internet site, and the list of identity criteria would include identity by comparing `URL` values. If the user does not choose a World-Wide Web collection, these choices will not be presented. SenseMaker is able to tailor these choices by checking with the individual collections to find out what attributes they support natively, determining how to map each of these native attributes onto the chosen canonical attribute model, and then using knowledge about what attributes are present to pare down the chosen canonical attribute model to just those attributes that are relevant. Note that the current mapping from native attributes onto the chosen attribute model is performed entirely within SenseMaker.

SenseMaker further tailors the presented choices by utilizing a *structured* canonical attribute model. A structured canonical attribute model makes relationships among attributes explicit. The SenseMaker structured canonical attribute model encodes generalization and composition relationships. For example, the model records that a `Reporter` is a kind of `Creator`, and that a `Publication Date` is made up of a `Publication Day`, `Publication Month`, and `Publication Year`. This relationship information allows SenseMaker to present the user with choices that are at the right level of granularity for the given set of results. If a user chooses only newspaper databases, then bundling by `Reporter` is offered as a possibility. If a user chooses both a newspaper database and a book database, then bundling by `Creator` subsumes bundling by `Reporter`—thus allowing books and newspaper articles by the same person to be bundled together.

Our experience with SenseMaker confirms the need for the requirements discussed in the section on query formulation (Section 2.2). For the SenseMaker approach to scale, we need to include attribute models and attribute translators in our architecture, and we need to develop protocols for communicating with these entities. Furthermore, the SenseMaker design adds two additional requirements:

- *We need to include structured attribute models as first-class objects and to establish methods for extracting the structure information from these models.*
- *We need attribute model translators that are capable of translating from native attributes and native attribute values into canonical attributes and canonical attribute values.*

3 The InfoBus Metadata Architecture

Motivated by the concrete metadata needs described in Section 2, we have designed an integrated metadata architecture. In this section, we begin by giving an overview of how our InfoBus works. Then we describe our metadata architecture and show how it fits into the InfoBus.

3.1 InfoBus Overview

Details of the Stanford InfoBus design are described in [1]. We give here only enough detail to provide context for our metadata architecture.

Our InfoBus consists of distributed objects that communicate with each other through remote method calls. In particular, we use the CORBA specifications [17], with Xerox PARC's ILU as the object system implementation [18].

Existing external services with different interfaces are made accessible by *service proxies*. These are objects that provide a standard set of methods on "one side," but can also communicate with the services they represent. For example, a proxy that represents Knight-Ridder's Dialog Information Service responds to the same methods as proxies that represent World Wide Web services (such as AltaVista or Lycos). Proxies shield InfoBus clients from the differences in access medium and protocol.

Figure 4 illustrates the InfoBus infrastructure. It shows some of the different services that co-exist on the InfoBus, including InfoBus-specific services, user interface services, information processing services, and information sources. Examples of InfoBus-specific services include *GlOSS*, the resource discovery service discussed in Section 2.1, and the query translation service discussed in Section 2.3. User interface services that we have developed include DLITE (Section 2.2) and SenseMaker (Section 2.4). The InfoBus proxy strategy allows us also to include external interfaces (such as a Z39.50 interface) on the InfoBus. Likewise, the InfoBus can use proxies to accommodate information processing services (such as a document summarization service) and personal information sources (such as an e-mail archive). This paper focuses primarily on the metadata needs of our InfoBus-specific services and user interface services. However, we have purposefully made our architecture extensible so that we can address the metadata needs of other non-search-related services at a later point.

Whenever possible, we try to present InfoBus components as collections of objects called `Item`. These are objects with various standard methods and arbitrary numbers of property-name/property-value pairs that may represent any real world object, ranging from a piece of text to a citation to a person. The property values are individually retrievable.

Search service proxies present themselves as instances of the class `ConstrainableCollection`. These represent sets of `Items` and respond to the method `ConstrainCollection()`, which returns a subset of the contained `Items`. For example, given a Knight-Ridder Dialog proxy `DP` and an Altavista proxy `AP`, both calls `DP.ConstrainCollection(query)` and `AP.ConstrainCollection(query)` return collections of `Items` that represent results.

Search service proxies may provide access to nested levels of subcollections that may be searched individually or together. Figure 5 shows an example.

In general, the structure of a proxy reflects the structure of the external service to which it provides access. When submitting a query to a `ConstrainableCollection`, any desired target subcollections are specified as well.

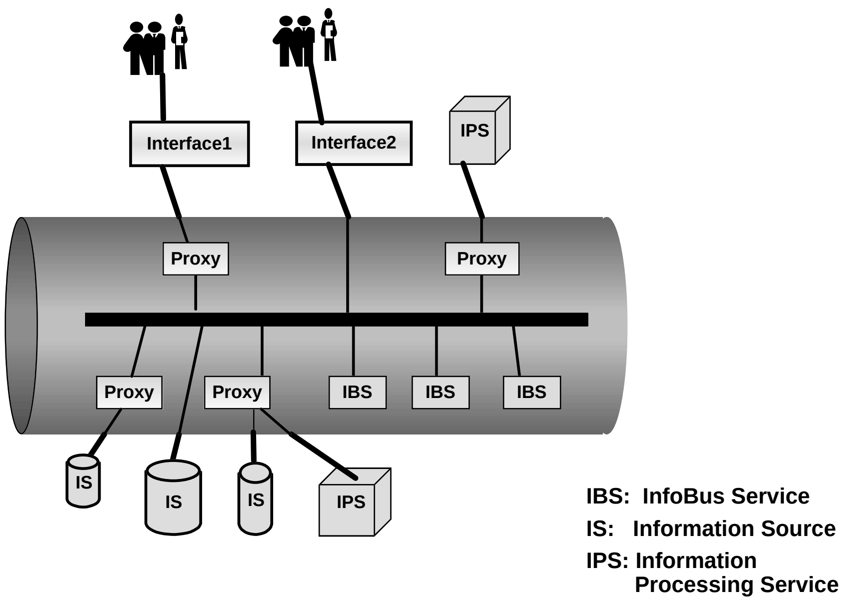


Fig. 4. The InfoBus Architecture

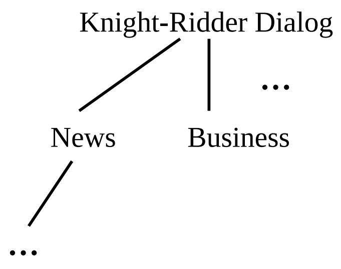


Fig. 5. The Dialog collection has several subcollections (e.g., News and Business) that may be searched individually or together.

3.2 InfoBus Metadata Facilities

Figure 6 presents our metadata architecture and shows how it fits into the InfoBus infrastructure. New components include *attribute model proxies*, *attribute model translation services*, a *metadata information facility* for each search service proxy, and a *metadata repository*.

An *attribute model proxy* represents a real world attribute model, just as a search service proxy represents a real world search service. For example, we might have one *attribute model proxy* for the USMARC set of bibliographic attributes (referred to as "fields" in the USMARC community) [19], another for the Dublin Core set of attributes [20], and so on. An *attribute model proxy* allows us to encapsulate information that is specific to an attribute model and independent of a search service proxy.

An *attribute model translation service* serves to mediate among the different metadata conventions that are represented by the *attribute model proxies*. These translation services, available via remote method calls, know how to translate (often approximately) attributes (and their values) from one attribute model into attributes from a second attribute model.

The *metadata information facility* that we attach to each search service proxy is responsible for exporting metadata about the proxy as a whole, as well as for exporting metadata about the collections to which it provides access. Collection metadata includes descriptions of the collection, declarations as to what attribute models are supported, information about the collection's query facilities, and the statistical information necessary for meta-searchers to predict the collection's relevance for a particular query.

Finally, the *metadata repository* is a local database that caches information from the other metadata facilities in order to produce a one-stop-shopping location for locally valuable metadata. We allow for the *metadata repository* to pull metadata from the various facilities, as well as for the facilities to push their metadata to the repository directly.

In the following sections, we describe each of these facilities in more detail. Several are implemented in our current prototype testbed; others are still under construction. Overall, our goal in implementing this architecture is to allow our InfoBus services to progress from

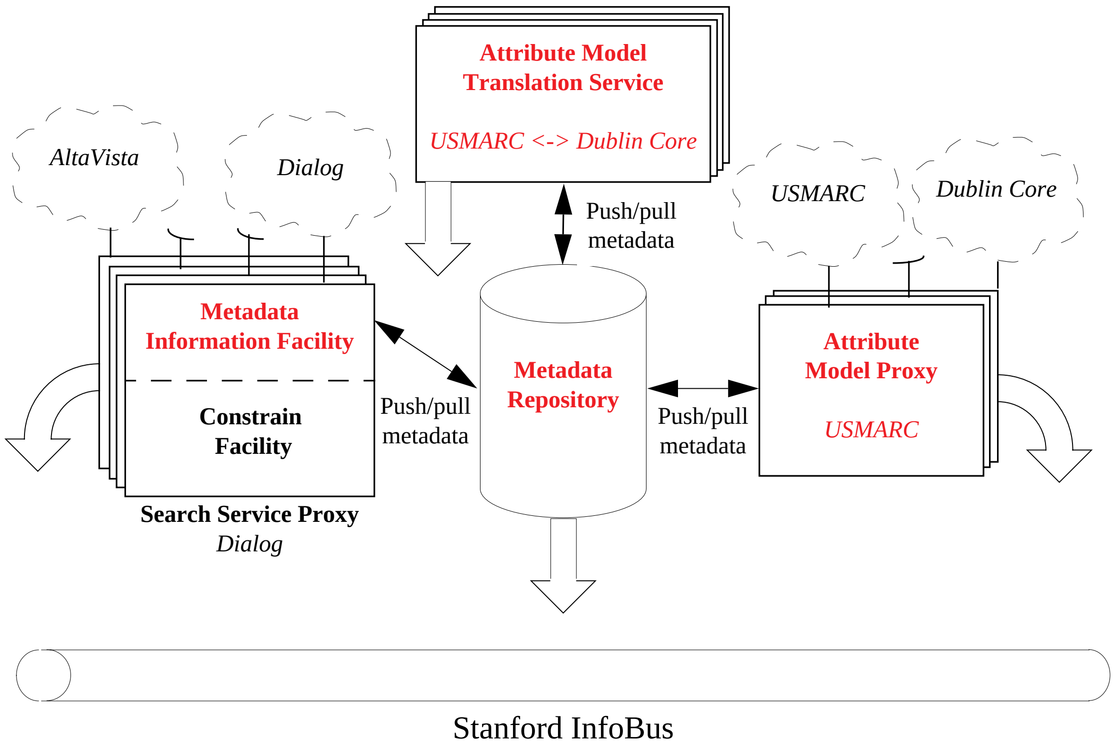


Fig. 6. Our Metadata Architecture

ad-hoc metadata interactions to well-specified, general, and scalable metadata interactions.

3.2.1 Attribute Model Proxies

Attribute model proxies are our way of making attribute models first-class objects in our computational environment. They allow us to store and search over attribute-specific information. On a per-attribute basis, attribute model proxies maintain attribute documentation and declare attribute value types. Furthermore, attribute model proxies record what relationships hold among the included attributes. This latter feature allows us to represent attribute models that are more complex than flat name spaces.

In the InfoBus ontology, an attribute model proxy is a `ConstrainableCollection` that contains `AttributeItem` instances. The fact that the attribute model proxy is a `ConstrainableCollection` means that it is accessible via the same interface as all other search service proxies. In other words, the attribute model proxy has a `ConstrainCollection()` method that responds to a query by returning the appropriate subset of the included `AttributeItem`s.

Each of the `AttributeItem`s that make up an attribute model proxy contains information that is relevant for a particular attribute, independent of the capabilities possessed by any particular search service proxy. Specifically, an `AttributeItem`'s properties include the following:

- `attrModelName: String`
- `attrName: String`
- `attrValueType: String`
- `attrDocumentation: String`

The `attrModelName` and `attrName` are both strings that serve to identify the `AttributeItem` uniquely. The `attrModelName` is repeated in all items to make them self-contained. This is important when the items are passed around the system to components other than the attribute model proxy. For example, meta-information recorded for a particular book might include values for attributes from many attribute models, including USMARC [19], Dublin Core [20], Z39.50 Bib1 [21], and one of the Stanford structured attribute models.

The `attrValueType` dictates the data type for the `AttributeItem`'s values. We use the interface specification language that is part of our CORBA implementation to specify these types. It is up to each search service proxy to ensure that the values it returns conform to these type specifications. If the external service that the proxy represents natively returns a different type, then the proxy is expected to transform the value into the specified type before returning it.

For example, let us assume a simple, flat attribute model that includes `Title`, `Individual Author`, and `Corporate Author`. A proxy for this model is a `ConstrainableCollection` containing the following `AttributeItem` instances:

```
SimpleModel: {
  inst1: {
    attrModelName -> 'Simple-Model'
    attrName -> 'Title'
```

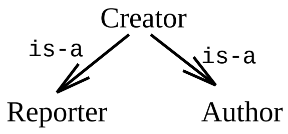


Fig. 7. A complex attribute model with the "is-a" relationship among its attributes.

```
    attrValueType -> 'String'
    attrDocumentation -> 'This is a title for
                  an entire volume of work.'}

  inst2: {
    attrModelName -> 'Simple-Model'
    attrName -> 'Individual Author'
    attrValueType -> 'SEQUENCE of String'
    attrDocumentation -> 'First-name/
                           last-name...'}

  inst3: {
    attrModelName -> 'Simple-Model'
    attrName -> 'Corporate Author'
    attrValueType -> 'Record {
                          String: companyName,
                          Integer: sicCode}'
    attrDocumentation -> 'Name as stated in
                   corporation records...'}}
```

Since `SimpleModel` is a `ConstrainableCollection`, any InfoBus component is free to search over instances of it. If `SM` is an instance of `SimpleModel`, then executing the statement `authorAttrs = SM.Constrain("attrName Contains Author")` will result in setting `authorAttrs` to a list of `inst2` and `inst3`. Finding out details about the two kinds of `Author` attributes is then as simple as examining the properties of `inst2` and `inst3`.

In this example, `SimpleModel` is a flat attribute model: all of the attributes are independent of each other. However, we saw in Section 2.4 that some user interface services require *structured* attribute models. Figure 7 depicts a structured attribute model that encodes "`is-a`" relationships among its attributes. In our architecture, the attribute model proxy for this attribute model would be a `ConstrainableCollection` containing `AttributeItems` subclassed to include `is-a()` methods. (Other relationships between the attributes in an attribute model, like "`has-a`," can be treated analogously.)

A search service proxy that supports this attribute model could use its relationship information when processing queries. For example, a search service proxy might determine that items match the query `Creator contains Ullman` if they contain "Ullman" in the value of the `Creator` attribute or if they contain "Ullman" in the value of descendant attributes (i.e., in the value of `Reporter` or `Author`).

Thus, we see that attribute model proxies play an important role in our metadata architecture. Attribute model proxies allow InfoBus components to search over attribute models, to obtain documentation and value type information for specific attributes within an attribute model, and to obtain information about the relationships that hold among an attribute model's attributes. This is the functionality that SenseMaker needs (Section 2.4) in order to determine display attributes, bundling criteria, and identity criteria that are at the appropriate level of granularity. The attribute model proxies are also key to building the translation services that we describe next.

3.2.2 Attribute Model Translation Services

In heterogeneous environments, many different attribute models co-exist together. Therefore, we need to have strategies for resolving the inevitable mismatches that will arise when InfoBus components that support different attribute models attempt to communicate with each other. For example, consider a bibliographic database proxy and a client of that proxy. The bibliographic database proxy might support only the Dublin Core attribute model, while the client might support only the USMARC bibliographic data attribute model. In order for this client and this proxy to communicate with each other, they must be able to translate from USMARC attributes to Dublin Core attributes and vice versa. In other words, they require intermediate attribute model translation services. Of course, in some cases, translation may not even be possible at all: consider translating from an attribute model designed for chemistry databases into an attribute model designed for ancient Greek texts.

Even when translation is appropriate, the translation from one attribute model to another is often difficult and lossy. For example, the Dublin Core describes authorship using the single attribute `Author`. However, USMARC distinguishes among several different types of authors, including `Corporate Author` (recorded in the `100` attribute) vs. `Individual Author` (recorded in the `110` attribute). When translating an `AttributeItem` from Dublin Core to USMARC, a decision must be made whether to translate the Dublin Core `Author` attribute value into a USMARC `100` attribute value or a USMARC `110` attribute value. This may be hard-coded, or the translation may be performed heuristically and may take into account the other attribute values of the `Item` being translated.

Translation services do more than map source attributes onto target attributes. They must also convert each attribute value from the data type specified for the source attribute into the data type specified for the target attribute. This conversion can be quite complex. For example, one attribute model might call for authors to be represented as lists of records, where each record contains fields for first name, last name, and author address. Another model might call for just a comma-separated string of authors in last-name plus initials format. When translating among these values, some information may again be lost if, for example, the address is simply discarded.

Attribute model translation services are thus an integral component in our metadata architecture. They may be accessed by a variety of other InfoBus components, including search service proxies. For example, a search service proxy might choose to use attribute model translators to be attractive to more clients, because they can then advertise that they deal in multiple attribute models. On the other hand, clients might use attribute model translators in order to ensure their ability to communicate with a wide variety of search services. In addition to query translators (Section 2.3), the attribute model translation services are needed during query formulation (Section 2.2) to map the canonical attributes in the user interface to the actual attributes supported by the collections. Also, SenseMaker (Section 2.4) uses the attribute model translators to convert the items returned from the collections into items that use the canonical attribute model.

The methods on standard model translation services are:

- `getNameMapping(toAttrModel, fromAttrModel, fromAttrName): SEQUENCE of String`
  /* Given a `fromAttrModel` attribute name, returns the corresponding `toAttrModel` attribute names */
- `getValueMapping(toAttrModel, fromAttrModel, toAttrName, fromAttrName, fromAttrValue): Any`
  /* Given a `fromAttrModel` attribute name and value, plus the `toAttrModel` attribute to which these should be mapped, returns the corresponding `toAttrModel` value */
- `getToAttrModels(): SEQUENCE of String`
  /* Returns the list of `toAttrModels` that this translator accepts */
- `getFromAttrModels(): SEQUENCE of String`
  /* Returns the list of `fromAttrModels` that this translator accepts */
- `getDocumentation(): String`
  /* Returns human-readable documentation for this translator */

For heuristic translators, we can subclass and add an additional parameter called `item` to the first two methods. These heuristic translators optionally allow passing an item pointer, so that the translator can use knowledge about all of the `item`'s attribute values to drive its translation algorithm. Our current implementation includes translators that perform non-heuristic mappings between two attribute models, and translations to and from a structured, hierarchical model. The implementation includes heuristic translators from USMARC to BibTeX, and Refer to BibTeX models.

3.2.3 Search Service Proxy Metadata Information Facilities

Each service proxy exports metadata information about itself and about the collection to which it provides access. Initially, clients can use this information to make judgments about how well the search service matches its needs. Later, clients can use the information to determine how best to access the collection maintained by the search service (i.e., what capabilities the search service supports).

We have decided to make the interface for accessing the metadata facility of search service proxies very simple in order to encourage proxy writers to provide this information. Search service proxy metadata is accessed via the call `getMetadata(subCollectionName)`. For each subcollection supported by the proxy, this call returns two metadata objects. Alternatively, each proxy may opt to "push" these metadata objects to its clients. The first metadata object (Table 1) contains the general service information, and it is based heavily on the *source metadata* objects defined by STARTS [22]. The general service information includes human-readable information about the (sub)collection, as well as information that is used by our query translation facility. Examples for the latter are the type of truncation that is applied to query terms, and the list of stopwords. Our current translation engine is driven by local tables containing this kind of information about target sources.

The `collectionName` attribute is the name of the (sub)collection that the metadata object describes. In case of a subcollection, the `parentCollectionName` attribute is the name of the immediately enclosing collection. Finally, if the (sub)collection described is in turn a `ConstrainableCollection` with subcollections, the `subCollectionNames` attribute lists its subcollections. For an example, consider the Dialog collection as depicted in Figure 5. The metadata object for the Dialog collection lists "Dialog" as the `collectionName`, no `parentCollectionName`, and "News" and "Business" as the `subCollectionNames`. The metadata object for the Business subcollection has "Business" as its `collectionName`, "Dialog" as its `parentCollectionName`, and no `subCollectionNames`.

Another interesting attribute of our first metadata object is `contentSummaryLinkage`. (See Table 1.) The value for this attribute is a URL that points to a *content summary* of the collection. Content summaries are potentially large, hence our decision to make them retrievable using `ftp`, for example, instead of using our protocol. The content summary follows the STARTS [22] content summaries, and consists of the information that a resource-discovery service like *GlOSS* needs (Section 2.1). Content summaries are formatted as Harvest SOIFs (`http://harvest.transarc.com/afs/transarc.com/-public/trg/Harvest/user-manual/`).

Example 3. Consider the following content summary for a collection:

```
@SContentSummary{
Version{10}: STARTS 1.0
Stemming{1}: F
StopWords{1}: F
CaseSensitive{1}: F
Fields{1}: T
NumDocs{3}: 892

Field{5}: Title
DocFreq{11023}: "algorithm" 53
                "analysis" 23
...
```

Metadata Attribute	*Description*
`version`	Version of the metadata object
`collectionName`	Name of the (sub)collection being described
`parentCollectionName`	Name of the immediate parent collection, if any
`subCollectionNames`	Name of the available subcollections
`attrModelNames`	Attribute models supported
`attrNames`	Attributes supported
`booleanOps`	Boolean operators supported
`proximity`	Type of word proximity supported
`truncation`	Truncation patterns supported
`implicitModifiers`	Modifiers implicitly applied (e.g., stemming)
`stopWordList`	Stopwords used
`languages`	Languages present (e.g., `en-US`, for American English)
`contentSummaryLinkage`	URL of the content summary of the collection
`dateChanged`	Date the metadata object last changed
`dateExpires`	Date the metadata object will be reviewed
`abstract`	Abstract of the collection
`accessConstraints`	Constraints for accessing the collection
`contact`	Contact information of collection administrator

Table 1. The attributes present in the first metadata object for a (sub)collection, describing general service characteristics.

```
Field{6}: Author
DocFreq{1211}: "ullman" 11
               "knuth" 15
...
}
```

This summary indicates that the word "algorithm" appears in the `Title` of 53 items in the collection, and "ullman" as an `Author` of 11 items in the collection, for example.

The second metadata object returned by the `getMetadata()` method contains attribute access characteristics. This is attribute-specific information for each attribute that the proxy supports. Recall that attribute model proxies contain only information that is independent from any particular search services. Attribute access characteristics complement this information in that they add the service-specific details for each attribute.

For example, some search services allow only phrase searching over their `Author` attributes, while others allow keyword searching. Similarly, some search services may index their publication attributes, while others may not. The attribute access characteristics describe this information for each attribute supported by the (sub)collection specified in the `getMetadata()` call. The query translation services (Section 2.3) need this information to submit the right queries to the collections.

Notice that this design does not allow clients to query search service proxies directly for their metadata. Search service proxies only export all their metadata in one structured "blob." The reason for this is that the InfoBus includes many proxies, and more are being constructed as our testbed evolves. We therefore want proxies to be as light weight as possible. Querying over collection-related metadata as exported by search service proxies is instead available through a special component, the metadata repository.

3.2.4 Metadata Repository

The metadata repository is a central, though possibly replicated, database of the system's metadata. The repository collects or subscribes to the metadata available from selected attribute model proxies, attribute model translation services, search service proxies, and other InfoBus services. The intent is for this repository to be a local resource for finding answers to metadata-related questions and for finding specialized metadata resources. The interface to the repository is again the same as a search service proxy interface. Subcollections within the repository include:

- A subcollection containing all `AttributeItem`s from locally relevant attribute model proxies. This subcollection can, for example, be used to search for documentation on Dublin Core attributes.
- A subcollection containing locally relevant attribute model translator information, structured as `Item`s. This subcollection is useful for searching for translators to or from particular models. Searches return pointers to the translator components.
- General service information for locally relevant search service proxies. These `Item`s are obtained through `getMetadata()` calls on the proxies, as discussed in the previous section. This subcollection can be used, for example, to find collections whose abstracts match a user's information need. It can also be used for more technical inquiries, such as for finding search proxies that support a given attribute model, or proxies that support proximity search.
- The attribute access characteristics of the locally relevant search proxies are also collected in the metadata repository. These are primarily useful to the query translator. Translators can, for example, find out which proxies support keyword searchable Dublin Core `Author` attributes.

The metadata repository provides the list of available collections to a resource-discovery service like *GlOSS*

Access Characteristic	*Description*
`collectionName`	Name of the (sub)collection being described
`attrModelName`	Model of the attribute
`attrName`	Name of the attribute
`searchRetrieve`	Whether the attribute is searchable, retrievable, or both
`modifierCombinations`	Legal combinations of modifiers (e.g., `stemming`, `>`) for the attribute

Table 2. The properties for one search attribute, as described in the second metadata object for a (sub)collection, and listing the attribute access characteristics.

(Section 2.1). It also lets the query formulation and translation services find out efficiently what attribute models and individual attributes are supported by each collection, for example (Sections 2.2 and 2.3).

3.2.5 Metadata Facilities Summary

In this section, we have seen that each metadata facility in our architecture serves to satisfy a number of the requirements imposed by our motivating set of InfoBus services. Thus, our metadata architecture is an integrated solution to the metadata needs of these independently developed and maintained services. We summarize here the matches between metadata facilities and requirements.

Attribute Model Proxies

- *They make attribute models first-class objects so that we can communicate with them and perform necessary transformations (Section 2.2).*
- *They allow attribute models to be structured and they establish methods for extracting the structure information from these models (Section 2.4).*

Attribute Model Translation Services

- *They translate canonical attributes and canonical attribute values into native attributes and native attribute values (Section 2.2).*
- *They translate native attributes and native attribute values into canonical attributes and canonical attribute values (Section 2.4).*

Search Service Proxy Metadata Information Facilities

- *They export the content summary associated with each collection (Section 2.1).*
- *They provide facilities for finding out efficiently what attribute models and individual attributes are natively supported by each search service (Section 2.2).*
- *They declare if each attribute is searchable and/or retrievable, and what the supported search methods are for searchable attributes (i.e., legal combinations of operators) (Section 2.3).*
- *They declare what operators are supported in addition to Boolean operators (e.g., the type of proximity operators supported) (Section 2.3).*
- *They export the stopwords used (Section 2.3).*
- *They export the vocabulary of the collection, i.e., the set of words indexed by the service, to emulate stemming when it is not natively supported (Section 2.3).*
- *They declare the details of other features, e.g., for truncation, the supported truncation patterns (Section 2.3).*

Metadata Repositories

- *They provide the list of locally available collections (Section 2.1).*
- *They provide facilities for finding out efficiently what attribute models and individual attributes are natively supported by each search service (Section 2.2).*

4 Related Work

In a distributed library environment, we can distinguish between metadata for information objects (e.g., documents) and metadata for information services (e.g., search services). The encoding of metadata for information objects can facilitate the unification of heterogeneous information objects, while the encoding of metadata for information services can facilitate communication among disparate services. Since interoperability is an important goal of our InfoBus, we have designed a metadata architecture that encompasses both types of metadata. In this section, we discuss related work on metadata for information objects and metadata for information services.

Many recent efforts have concentrated on the metadata for information objects. In this context, the term metadata refers to the description of information objects to support the major functions of digital libraries such as search, assessment, and acquisition of information. Work in this area generally falls into two categories: specification of metadata sets and architectures that integrate them.

Relevant work in the first category (specification of metadata sets) includes, for instance, the Bib-1 attribute set in Z39.50 and the Dublin Core. The Z39.50 Bib-1 attribute set [23, 21] registers a large set of bibliographic attributes. The focus of the Dublin Core [20] is primarily on developing a simple yet usable set of attributes to describe the essential features of networked documents (e.g., World-Wide Web documents), which the report of the Dublin meeting terms “document-like objects.” The Dublin Core metadata set consists of 13 metadata elements, including familiar descriptive attributes such as `Author`, `Title`, and `Subject`.

These standard metadata sets fit nicely into our architecture. As we have described, we represent metadata sets by attribute models that serve as reference points for defined attributes. In addition, our attribute models can be more structured than just flat name spaces. Our attribute models go beyond these metadata sets because they can optionally include structure and because they are reified as searchable collections.

Given the existence of various metadata sets for information objects, and the fact that no single set covers all the possible aspects, there have also been significant efforts in developing architectures that support the integration and interoperation of various metadata sets. Work on this aspect of the metadata problem includes the Warwick Framework [24], and the Jet Propulsion Laboratory's DARE metadata model [25].

The Warwick Framework proposes a container architecture as a mechanism for incorporating attribute values from different metadata sets in a single information object. Within each object are "metadata packages," one for each distinct metadata set, e.g., Dublin Core or USMARC. The Warwick Framework also includes implementation suggestions for integrating it with HTML, MIME, SGML, and distributed object technology (e.g., CORBA).

The Warwick Framework is essentially an encoding scheme that incorporates various metadata packages with document data. Common to our work is the support of multiple metadata sets in describing documents. However, as we represent documents as a flat set of attribute-value pairs (similar to HTML), the encoding scheme of the Warwick Framework is more sophisticated because its structure supports recursion and indirection. Consequently, to fully express the Warwick Framework we would need to extend our document models. On the other hand, our work complements the Warwick Framework in that we do provide attribute models as registries for metadata attributes. The procedure for specifying and registering new metadata sets is still an open issue in the Warwick Framework.

The DARE metadata model was developed to support the interaction with metadata elements and to tie these elements to data themselves. It uses ODL (Object Definition Language) to define new metadata elements that are stored in a "data dictionary." This notion of data dictionaries is similar to our attribute models. However, to facilitate interoperability, we also provide translation services that translate attributes among different models.

In addition to the work on metadata for information objects discussed above, there are also proposals that support metadata for search services. In this context, the efforts that are probably closest to what is described here are the Explain facility of Z39.50-1995 (i.e., Version 3 of Z39.50 [23]) and Stanford's STARTS [22], both of which require services to export their "source metadata." The former represents a standard effort for information retrieval while the latter is intended to be an informal, lightweight agreement for interoperability among Internet search engine vendors.

The Explain facility is the primary mechanism for Z39.50 clients to discover servers' capabilities. Explain-based clients can dynamically configure themselves to match individual servers so that they can support more than the least common denominator of the servers. Z39.50 servers present metadata about their services via the Explain facility. This metadata is essentially another database that can be queried by the clients via the Z39.50 protocol. In the Explain database, server characteristics are divided into categories, and database record structures are defined for each category. The GILS profile for Z39.50 [26] defines a metadata attribute set for search services.

The Explain facility provided by Z39.50 servers corresponds to the metadata general information facility supported by our service proxies. The major distinction is that the interface to access the service metadata provided by our proxies is simpler than that of the Explain facility. As explained in the previous sections, we have decided to shift the more complex functionalities (e.g., searchable metadata) to the metadata repositories. The rationale is to make proxies as lightweight as possible and therefore easy to implement. As every service needs a proxy to enter our InfoBus, making proxies lightweight becomes critical. On the other hand, this simplicity may not be an issue for Z39.50 as both their clients and servers will necessarily have most of the required capabilities [27]. However, our architecture can benefit from the Explain facility; it should be relatively easy to build proxies to Explain-compliant services that will support our proposed metadata facility.

Finally, another relevant effort on top of which we built our architecture is STARTS [22] (`http://www-db.stanford.edu/~gravano/starts.html`). STARTS is an informal "standards" effort coordinated by Stanford, whose main goal is to facilitate the interoperability of search engines for text. STARTS specifies what metadata should be exported by each collection of text documents. This information is essentially what the search-service proxies export through their metadata information facility (Section 3.2.3). STARTS does not describe, though, a more sophisticated metadata architecture. It just specifies the information that the collections should provide. As with the Explain facility, the architecture that we present in this paper benefits from STARTS-compliant services: it is easy to build proxies for such services that will satisfy our metadata requirements.

Notice that unique in our architecture is the metadata repository that serves as a central, though possibly replicated, database of metadata. The advantages are, first, that the clients may conveniently use it as a local cache of metadata. Second, it naturally serves as a service "directory" so that interesting queries may be made to help in resource discovery, e.g., `find all services that support the Dublin Core Title attribute`. Finally, service proxies can "publish" their metadata to the metadata repositories. This makes the update of service metadata easier to propagate in the information space.

5 Conclusion

In this paper, we have surveyed the metadata needs of our InfoBus services and presented a metadata architecture that meets these needs. In the process, we have outlined a framework for understanding different classes of metadata and metadata uses. The components of our proposed architecture are integrated into the InfoBus. Each component is a distributed object that can communicate through remote method calls. Furthermore, the metadata repository is an InfoBus `ConstrainableCollection` object that can be searched using our already established search protocol.

More specifically, the components in our metadata architecture include: attribute model proxies, attribute model translation services, metadata information facilities for InfoBus search service proxies, and local metadata repositories. Together, these components can provide, exchange, and describe metadata for information objects and metadata for information services. We have found that these facilities are necessary building blocks for scalable interoperability.

Our current implementation includes six attribute model translators, table-driven query translation, content statistics over the CS-TR collection, and dynamic attribute derivations. We are currently modularizing these facilities, building attribute model proxies, adding the metadata repository, and making our search proxies compliant. The pieces of the architecture that are already implemented allow for interoperability among the services and collections that are in the InfoBus today. Once our architecture is complete, we will have a framework that enables us to incorporate many more heterogeneous services and collections into the InfoBus. In addition, we will be able to extend it to meet the needs of other types of information processing services.

References

1. Andreas Paepcke, Steve B. Cousins, Héctor García-Molina, Scott W. Hassan, Steven K. Ketchpel, Martin Röscheisen, and Terry Winograd. Towards interoperability in digital libraries: Overview and selected highlights of the Stanford Digital Library Project. *IEEE Computer Magazine*, May 1996.
2. Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke. Metadata for digital libraries: Architecture and design rationale. Technical Report SIDL-WP-1997-0055, Stanford University, 1997. Accessible at `http://www-diglib.stanford.edu/cgi-bin/WP/get/-SIDL-WP-1997-0055`.
3. Andreas Paepcke. Searching is not enough: What we learned on-site. *D-Lib Magazine*, May 1996.
4. Andreas Paepcke. Information needs in technical work settings and their implications for the design of computer tools. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 5:63–92, 1996.
5. Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, May 1994.
6. Luis Gravano and Héctor García-Molina. Generalizing *GlOSS* to vector-space databases and broker hierarchies. In *Proceedings of VLDB '95*, pages 78–89, September 1995.
7. Steve B. Cousins, Scott W. Hassan, Andreas Paepcke, and Terry Winograd. A distributed interface for the digital library. Technical Report SIDL-WP-1996-0037, Stanford University, 1996. Accessible at `http://www-diglib.stanford.edu/-cgi-bin/WP/get/SIDL-WP-1996-0037`.
8. Steve B. Cousins. A task-oriented interface to a digital library. In *CHI 96 Conference Companion*, pages 103–104, 1996.
9. Michelle Q Wang Baldonado and Steve B. Cousins. Addressing heterogeneity in the networked information environment. *Review of Information Networking*, to appear.
10. D. T. Hawkins and L. R. Levy. Front end software for online database searching Part 1: Definitions, system features, and evaluation. *Online*, 9(6):30–37, November 1985.
11. M. E. Williams. Transparent information systems through gateways, front ends, intermediaries, and interfaces. *Journal of the American Society for Information Science*, 37(4):204–214, July 1986.
12. Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, August 1996.
13. Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Predicate rewriting for translating boolean queries in a heterogeneous information system. Technical Report SIDL-WP-1996-0028, Stanford University, 1996. Accessible at `http://www-diglib.stanford.edu`.
14. M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
15. J. B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11(1-2):22–31, 1968.
16. Michelle Q Wang Baldonado and Terry Winograd. SenseMaker: An information-exploration interface supporting the contextual evolution of a user's interests. In *Proceedings of CHI 97*, 1997.
17. Object Management Group. The Common Object Request Broker: Architecture and specification. Accessible at `ftp://-omg.org/pub/CORBA`, December 1993.
18. Doug Cutting, Bill Janssen, Mike Spreitzer, and Farrell Wymore. *ILU Reference Manual.* Xerox Palo Alto Research Center, December 1993. Accessible at `ftp://-ftp.parc.xerox.com/pub/ilu/ilu.html`.
19. USMARC format for bibliographic data: Including guidelines for content designation, 1994.
20. Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel, Jr. OCLC/NCSA metadata workshop report. Accessible at `http://www.oclc.org:5047/oclc/research/publications/-weibel/metadata/dublin_core_report.html`, March 1995.
21. Z39.50 Maintenance Agency. Attribute set Bib-1 (Z39.50-1995): Semantics. Accessible at `ftp://ftp.loc.gov/pub/-z3950/defs/bib1.txt`, September 1995.
22. Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford protocol proposal for Internet retrieval and search. Technical Report SIDL-WP-1996-0043, Stanford University, August 1996. Accessible at `http://www-diglib.stanford.edu/cgi-bin/WP/get/-SIDL-WP-1996-0043`.
23. National Information Standards Organization. *Information Retrieval (Z39.50): Application Service Definition and Protocol Specification (ANSI/NISO Z39.50-1995).* NISO Press, Bethesda, MD, 1995. Accessible at `http://lcweb.loc.gov/-z3950/agency/`.
24. Carl Lagoze, Clifford A. Lynch, and Ron Daniel, Jr. The Warwick Framework: A container architecture for aggregating sets of metadata. Technical Report TR96-1593, Cornell University, Computer Science Dept., June 1996.
25. Jason J. Hyon and Rosana Bisciotti Borgen. Data archival and retrieval enhancement (DARE) metadata modeling and its user interface. In *Proceedings of the First IEEE Metadata Conference*, Silver Spring, Maryland, April 1996. IEEE.

26. Government Information Locator Service (GILS), 1996. Accessible at `http://info.er.usgs.gov:80/gils/`.
27. Denis Lynch. Implementing Explain. Accessible at `ftp://-ftp.loc.gov/pub/z3950/articles/denis.ps`.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.