An Architecture for Automatic Reference Linking

Donna Bergmark and Carl Lagoze
Cornell Digital Library Research Group*

Abstract

Along with the explosive growth of the Web has come a great increase in on-line scholarly literature. More and more researchers are turning to their computers for the latest results in their field. Not only is Web retrieval usually faster than a walk to the library, but the information obtained from the Web is generally more current than what appears in printed publications.

The increasing proportion of on-line scholarly literature makes it possible to implement functionality desirable to all researchers – the ability to access cited documents immediately from the citing paper. Implementing this direct access is called "reference linking".

While many authors insert explicit hyperlinks into their papers to support reference linking, it is by no means a universal practice. The approach taken by the Digital Library Research Group at Cornell employs *value-added surrogates* as a generalizable mechanism for providing reference-linking behavior in Web documents. Given the URL of an on-line document, a surrogate object is constructed for that paper, which then processes document's content to extract reference linking data. The surrogate then exposes the reference linking data through a well-defined API, permitting the construction of reference linking services by external clients.

We present two examples of the many possible reference linking applications buildable on this API. We also describe a metric that measures the API's performance; currently we are (automatically) extracting reference linking information from HTML papers with more than 80% accuracy.

Keywords: reference linking, OpCit, value-added surrogates, HTML, data extraction

*NSF Grant # IIS-9907892

1 Background and Motivation

The variety of on-line scholarly literature has grown along with the web. Informal on-line archives are repositories for papers and technical reports. Proceedings are increasingly often published on the Web. The collection of on-line journals is growing. More and more scholarly results are simply posted on personal or group Web sites. Thus the Web is becoming an up-to-date repository of information for the scientific researcher, and researchers are turning to their computers to keep current on results in their field. Not only is Web retrieval usually faster than a walk to the library, but information obtained from the Web is generally more current than what appears in printed publications.

The increasing proportion of on-line scholarly literature (nearly 10,000 on-line journals in 1999 [13]) makes it possible to implement a function desirable to all researchers – the ability to access cited documents immediately from the citing paper. Implementing this direct access is called "reference linking".

Reference linking is actually an old idea. Classical reference linking arose from a desire to study citation patterns among scholarly articles. The Science Citation Index[6] founded by Eugene Garfield in the 60's was invented to do just that, and was a spectacular success. It was, however, based on human labor. For every paper examined, the staff captured that paper's bibliographic data, and then went to the reference section and did the same for each reference there, or at least for those references to journals covered by the SCI.

As a result, one could look up links using the Science Citation Index and build a graph as shown in Figure 1. The nodes in this graph represent the scientific papers, and directional arcs have two contextual meanings. Outgoing arcs, with respect to a specific node, lead to *references* of that paper. Incoming arcs, with respect to a specific node, originate from that paper's *citations*. Thus from the graph in Figure 1, we can observe that Paper C has 4 references and that Paper A has two citations. The links in the graph are *explicitly* contained in the Science Citation Index.

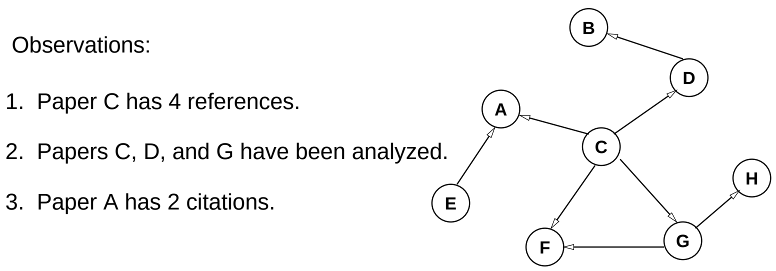

Figure 1: Classical Reference Linking

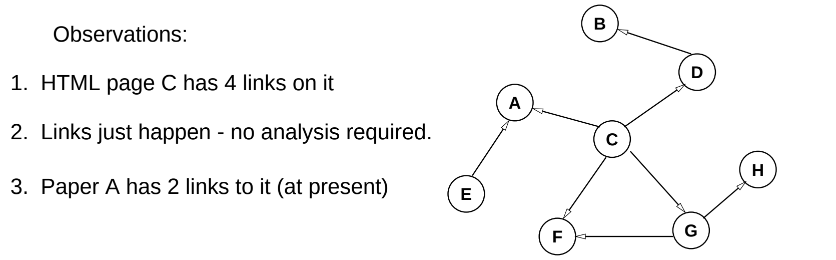

Figure 2: Linking on the Web

We then fast-forward some 40 years to the current time, where there is a growing amount of scholarly literature on-line. In many cases, the authors of this literature have assumed the classical reference taking task and have inserted references to other works on the Web. The use of URLs in many of these references allows efficient movement from the referencing to the cited work. Maintaining an account of a paper's citations is a considerably more daunting task for which the uni-directional links on the Web offer no help.

Figure 2 shows a sub-graph of interlinked Web papers. From the fragment shown here, we can deduce that document C has four links in it to other Web documents and document A has at present two links to it. While the graph may explicitly be stated in HTML hyperlinks, authors often fail to insert explicit links to references and citations can be discovered only by exhaustive analysis of the Web.

Our reference linking project aims to bridge the classical view and what exists today on the Web. We wish to make the graph in Figure 2 explicit for selected repositories, as well as automatically supply additional links where possible. A variety of uses exist for explicit hyperlink graphs (see for example [5, 7]). This work creates a *reference linking layer* on the Web that provides sufficient data for a variety of value-added *reference linking applications*. Some of these applications may be targeted at human use, consisting of a user interface for navigating the reference linked graph. Others may be *middleware*, massaging data for use by other applications, such as to provide a citation database. Our main objective is to provide this layer *automatically*, without the direct participation from authors or publishers.

Figure 3 is an example of a human-oriented reference linking application that exploits the data from a reference linking layer.

Imagine sitting in front of a computer screen, reading Document A and you come across an intriguing reference: "...Mitchell's seminal work on thunks[10]." If there is a copy of this work somewhere on-line, the "[10]" can be made into a clickable link that allows the user to start fetching that copy while continuing to read the original paper. For example, a JavaScript popup window that looks something like the

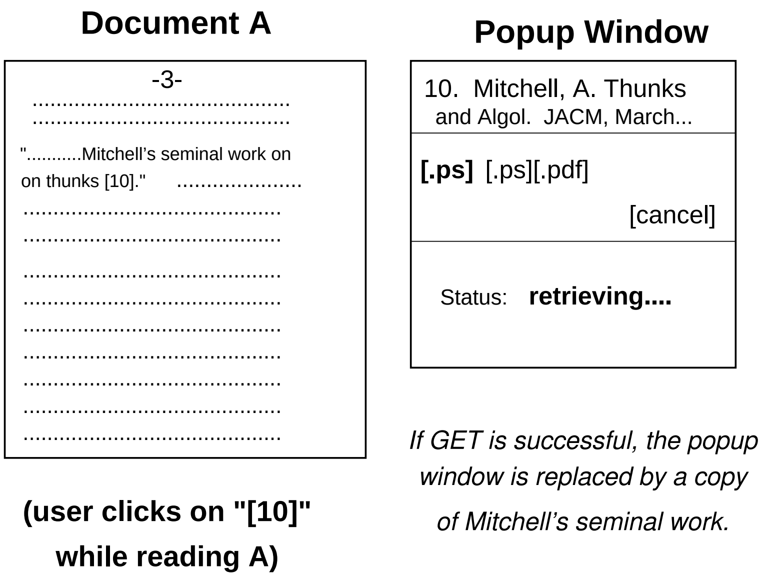

Figure 3: A Reference Linking Application

one on the right side of Figure 3 could display the complete reference string along with some choices of format (PostScript, PDF) in which the document might be retrieved; the user can retrieve one of these or cancel. Alternatively, clicking on the link could bring up a dialogue that presents choices to the user based on his/her access permissions to various on-line resources.

Implementing the functionality shown in Figure 3 requires a number of steps. The *reference linking layer* must:

1. Deduce that the string `[10]` in the text is a reference and that it matches the reference string, `10.  Mitchell, A. Thunks and Algol...`; in the reference section at the end of the paper.
2. Parse the reference string to decide what work it is, whether it is linkable, and whether it is something we've seen before so we can credit Mitchell with this paper as a citation, and – if we have seen it before while analyzing a repository – its URL.
3. Provide access to this reference linking data for use by applications.

The *reference linking application* must then:

4. Turn the "[10]" into a live link. In HTML and PDF this reference can be turned into a clickable anchor. For other formats an auxiliary display can provide clickable links.

From a data-flow perspective this process can be described as *analysis*, *data access*, and *presentation.* Our work has been concentrating on the first two. We describe an API that makes the data from this analysis available to client applications. In later work, we plan to examine the presentation layer.

1.1 Related Work

Reference linking is currently being implemented by a large group of publishers in the CrossRef project, which grew out of the DOI-X project described by Atkins [2]. This organization, consisting of professional journal publishers, is interlinking their on-line journals. Each publisher, using editorial input, generates its own metadata and shares it with the other publishers in the group.

Other systems, like ours, endeavor automatic metadata generation for Web items. For example, dc.dot `<http://www.ukoln.ac.uk/metadata/dcdot/>` extracts titles from <title> elements of HTML files, and keywords from headers and capitalized phrases. Jenkins [10] describes a system that automatically generates some Dublin Core metadata from specialized sources of data on the http server. Neither of these systems get their data by examining the body of the document itself, as we do. Also, our interest is only in bibliographic metadata, that which comprises citations and references. We do not collect keywords, nor do we attempt any classification of the papers that have been analyzed.

ResearchIndex[12] is a well-known reference linking application which seeks to build a large on-line database of computer science citation information, automatically. Their software is available, and we have borrowed some of their techniques. One significant thing we learned from their research is the potential value of retaining reference contexts [11]. Our work is distinguished from ResearchIndex by our clear architectural distinction between reference linking data and its presentation thus enabling a variety of reference linking services, while ResearchIndex's functionality is available only through their Web interface.

A related research project is context-sensitive reference linking, where the targets of clickable anchors depend upon the user's circumstances (i.e. which subscriptions does the user own, to what organization does the user belong). SFX[17], an advanced reference linking system that takes the user's context into account, was the first server for context-sensitive reference linking. Its developer, Herbert Van de Sompel, is now a member of the Digital Library Research Group at Cornell. We are actively discussing the merging of his work with the quite complementary work reported here.

Formally, our work is part of a larger project called OpCit based at Southampton University `<http://opcit.eprints.org>`. Funded jointly by JISC in the UK and the NSF Digital Libraries Initiative in the US, OpCit's goal is to interlink the online literature. The project is further described in [9]. The Open Journals project, OpCit's precursor, is discussed in [8].

2 Definitions

In this section we present basic terms and definitions that will be used in the remainder of the paper.

2.1 Items and Works

There are two different types of entities contained in Figure 3: there is Document A which the user is reading, and there is thing B, a work by Mitchell, which is referenced by A. There is a subtle, but important, difference between A and B. A is an *Item*, something that has a specific format, is on-line, and can be analyzed by a computer program. B is a *Work*, or an abstraction of a paper. In the example shown, B happens to exist in the form of several Items. In general, however, a Work need not be available on-line. Thus we say that Works exist in the form of zero or more on-line Items. (So, our definition of Item deviates slightly from standard library usage in that it is limited to on-line copies.) See [16, 15] for how the Work and Item distinction has played a role in traditional library cataloging theory.

In the rest of this report, we drop the capitalization of work and item, but continue to distinguish between the two terms, because the distinction is important.

2.2 References and citations

References and Citations were discussed with Figure 1 and their definitions need not be repeated here except to note that both references and citations exist in the context of works. In the remainder of this paper we will use four terms associated with references:

- *linkable references* are copies of works that can be found on-line. They are items as well as works.
- *reference anchor* will refer to the reference in the text of the paper being analyzed, e.g. "`[10]`" in Figure 3;
- *context* will be the sentence containing that reference anchor;
- *reference string* will be the complete description of the work as it occurs in the document (usually in a bibliography), e.g. "`10. Mitchell. A. Thunks ...`".

2.3 Repository

A *repository*, also known sometimes as an *archive*, is any collection of on-line items, e.g. an on-line journal, a department's technical reports, or a person's on-line bibliography.

2.4 Intralink and Interlink

If we analyzed every item in a repository, then we can *intralink* that repository, since if one item references a work that is itself an item in the repository, it is a linkable reference. We know its address.

Analyzing several repositories allows the *interlinking* of items in these repositories. The extreme limit of this work, as more and more items are analyzed, is to interlink the Web, at least the scholarly side of it.

3 The Reference Linking API

Our reference linking architecture is unusual in several respects. First, because we are aiming to link on-line literature without author or publisher intervention, we take an *automatic* approach to parsing documents in order to extract the item's metadata as well as the metadata of the item's references. In a recent article, Arms[1] argues that "automated librarianship", using computers to do some of the organizational tasks formerly done by humans, is necessary in order to effectively deal with costs of organizing and managing on-line content. Similarly, automated reference linking complements and extends that manually done by authors. The advantage of automatic metadata extraction, although it probably is less accurate than hand-generated metadata, is that it becomes possible to process a vastly larger set of Web content.

Secondly, we have a different approach to collecting and storing reference linking data. Most other reference linking projects (e.g. Open Journals and ResearchIndex) use tables to store information about works and items. For example, there might be one database of all the titles, and perhaps another database of all the authors, and a third with references. Some sort of API would be built on top of these databases to extract and present data as required.

Instead of using relational databases to store this information, we use item surrogates. A *surrogate* is a digital object that encapsulates reference linking information relating to one single item on the Web. Data is thus distributed across the collection of surrogate objects, and all the data relating to one item is grouped together within a single surrogate. A collection of permanently stored surrogates is not an object oriented database, but it would be natural to store the surrogates in one.

This use of surrogates for reference linking is consistent with our overall architectural approach to digital library research. We see "value-added surrogates"[14] as a vehicle for endowing digital objects with a wide variety of extensible behaviors (e.g. preservation, access management).

A third unusual aspect of our approach to reference linking is that we do not put together one monolithic reference linking service. Instead, we provide an API on top of which a variety of reference linking services can be built. Such an approach has been quite successful in other, unrelated endeavors (see, for example, [4]). We will present two applications that use this API.

3.1 How the API Works

The combination of surrogates and an API essentially allows us to access an on-line paper and ask it "what are your references" and "what is your metadata?" In fact, the API is a set of methods, where each *method* is simply one of the questions or requests that can be directed at the surrogate. Each surrogate answers the same set of questions, but with respect to its own item.

A typical use of the API would be to analyze all the papers in some repository or other collection (e.g. somebody's Web site). This architecture is depicted in Figure 4. The central column represents some repository of network-accessible documents. The items listed in this column are linkable (they are on-line) and therefore analyzable (their bits are available).

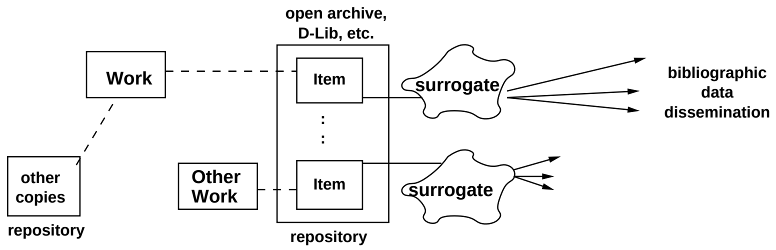

Figure 4: An Architecture for Reference Analysis

On the left are drawn the works that the items represent. Any work might have several copies spread across several repositories. All of these copies are "items" corresponding to that work.

To the right of the archive items are the surrogates, shown as "blobs". Their job is to perform the first of the three phases listed at the end of Section 1: analysis. As a result, they can disseminate bibliographic data about the item, and indirectly, about the work.

The surrogates can be created on-demand to supply the data, or they can be stored and used later to supply the data. Thus the reference linking layer imposed on Web objects can be static or dynamic or some mixture of both.

3.2 Components of the API

The six methods in our reference linking API are:

- `getLinkedText` – returns the contents of the paper (as data) augmented with reference linking information. One use of this method would be to display the document with some of its reference anchors turned into live links, as in Figure 3.
- `getReferenceList` – this interface would be used by applications wanting to know what references are contained in this paper. For example, if one were building the SCI, this would be the method to use, along with the next one.
- `getMyData` - this returns that paper's own metadata. It could have other uses; for example, a button labeled "get BibTeX" could invoke `getMyData` on the surrogate, and reformat the results into something suitable for cutting and pasting into a LaTeX bibliography.
- `getCurrentCitationList` – the list of works citing this paper to the best of the surrogate's knowledge. Support for citations is not immediately needed for reference linking, but is a valuable addition to any reference linking service, and so it is provided for in this API. This method is useful to client applications that want to know what other documents cite this one, as they might be related or provide more current information. If on-line, we have a *linkable citation*. Note that there is a subtle difference in tractability of making, for an item, its list of references and its list of citations. However, the more citations that are known for an item, the more explicit is its Figure 2 graph.
- `save` and `restore` - move a surrogate's local data to persistent storage for later retrieval.

3.3 Output from the API

This section provides details about the data returned by the API methods. Figure 4 showed the surrogates disseminating bibliographic information about their items, in response to a particular method in the API being invoked. Each method (except `save` and `restore`) returns a byte-stream of structured data coded in XML. For illustrative purposes, we look at the output of `getReferenceList` and `getLinkedText` in more detail.

The data disseminated by `getReferenceList` (Figure 5) contains elements in two XML namespaces; `dc` for the Dublin Core elements, and `api` for the elements specific to our API. The data is organized with one top level element, `<reference_list>`, which in turn consists of 0 or more `<reference>` elements. The example shown is for the second reference (`ord="2"`) in the surrogate's item.

The `<reference>` element consists of:

1. Bibliographic data related to the reference work, expressed in Dublin Core format.
2. The reference string exactly as it appeared in the item (enclosed in a `<literal>` element and entified).
3. Zero or more contexts in which the work was cited in this item, listed as `<context>`s within a `<context_list>`.[1]

Note the "[2]" in the example's `<context>`. This becomes a reference anchor in the text returned by `getLinkedText`. Should the surrogate's `getLinkedText` method be invoked, this reference anchor would be is enclosed in a `<reflink>` element. The middle of Figure 6 shows an example `<reflink>`. Note that 0 or more `<url>` subelements are allowed within a `<reflink>`.

The `<reflink>` elements can be translated (by a XSLT processor) into "actionable links", such as HREF's, XLinks, OpenURLS, or embedded JavaScripts. Figure 6 shows how the example `<reflink>` element might look after being converted into an XLink of type "simple".

3.4 A Java Implementation of the API

The API can be implemented in many languages. We briefly discuss our Java implementation in this section.

The API is implemented as three packages, only one of which (`Linkable.API`) is needed by reference linking applications. The other packages include one for parsing source documents (`Linkable.Analysis`) and another for helper routines (`Linkable.Utility`).

Only one parameter is required for constructing a surrogate object, the URL of the item to be parsed. The surrogate invokes one or another analyzer depending on the item's format (as determined by the item's filename extension). The first job of the analyzer is to convert the item to XML (XHTML). Formatting hints (e.g. font size, line breaks) are retained in the XML version to enable decomposition of the item into header, body, and reference sections.

After construction of the surrogate by the reference linking application, the item has been parsed and preliminary reference data has

[1]Zero contexts is a characteristic of papers that include gratuitous references.

```
<? xml version="1.0" ?>
<api:reference_list length="17"
      xmlns:api="http://www.cs.cornell.edu/cdlrg/..."
      xmlns:dc="http://purl.org/dc/elements/1.1/">
  <api:reference ord="1">
          :
          :
  <api:reference ord="2">
    <dc:title>
       Smart Objects, Dump Archives: A User-Centric,
       Layered Digital Library Framework
    </dc:title>
    <dc:date>1999-03-01</dc:date>
    <dc:identifier>10.1045/march99-maly</dc:identifier>
    <dc:creator>K Maly</dc:creator>
    <api:displayID>
       http://www.dlib.org/dlib/march99-maly/03maly.html
    </api:displayID>
    <api:literal tag="2.">
       Maly K, "Smart Objects, Dumb Archives: A User-Centric,
       Layered Digital Library Framework" in D-Lib Magazine,
       March 1999,
       &lt;http://www.dlib.org/dlib/march99-maly/03maly.html&gt;.
    </api:literal>
    <api:context_list anchor="[2]">
      <api:context>
         The need for standards to support the interoperation of
         digital library systems has been reported on before in
         D-Lib[1],[2] as have efforts to discover common ground
         in related standard processes(Dublin Core and INDECS[3]).
      </api:context>
    </api:context_list>
  </api:reference>
         :
         :
</api:reference_list>
```

Figure 5: XML for a Reference Object

Original Text:

```
... it was said [5] that ...
```

Linked Text with custom tag:

```
... it was said
<reflink ord="5" author="last-name-of-first-author"
title="title of this work"
year="1999">
<url>"http://www.some.org/filename"</url>[5]</reflink>
that ...
```

Linked Text with XLink:

```
... it was said
<ref-xl xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:type="simple"
xlink:href="http://www.some.org/filename">[5]</ref-xl>
that...
```

Figure 6: A linked XML Item, first with reflink then with XLink.

been stored into local data fields within the surrogate. Invoking one of the first four methods on the surrogate, e.g. `getLinkedText()`, causes further analysis of the reference data and culminates in an XML byte array. `Save` puts the surrogate's data into persistent storage, as an XML file. `Restore` retrieves this data and re-constructs the surrogate from it.

4 Experiments with the Architecture

The reference linking API can be used for a large variety of applications. This section briefly sketches two examples. The first example is an on-line application; the second is run off-line to produce stored data.

4.1 A Simple Display Application

The first example (Figure 7) shows how on-demand surrogate creation can be used as part of a reference linked document viewer, a useful overlay to the Web. This application, upon being given the URL of some on-line item, instantiates a surrogate object, passing it that URL. Instantiation of the surrogate causes the item at that URL to be analyzed. All further interactions with the reference linking API are via this surrogate.

Here is a sample snippet of Java code in this application:

```
Surrogate s = new Surrogate ( url );
display ( s.getLinkedText() );
```

This application uses the API to obtain the linked text for the item located at the specified url; the resulting XML byte array is then passed to a routine, `display()`, which presents the linked text to a user. The `<reflink>` elements have been translated into JavaScript, which handles full-text document retrieval for some reference anchors.

This example has shown the API being used in a dynamic mode.

4.2 Reference Linking the D-Lib Magazine

The second example gathers and stores reference linking information for future use. We used the Java implementation of the reference linking API to analyze D-Lib articles. D-Lib is an on-line journal that has been appearing eleven times a year since July 1995; it makes an excellent test bed for automatic extraction software because there is little editorial imposition of format on the papers published in the journal, and therefore provides a wide selection of paper layouts. All D-Lib articles are written in HTML.

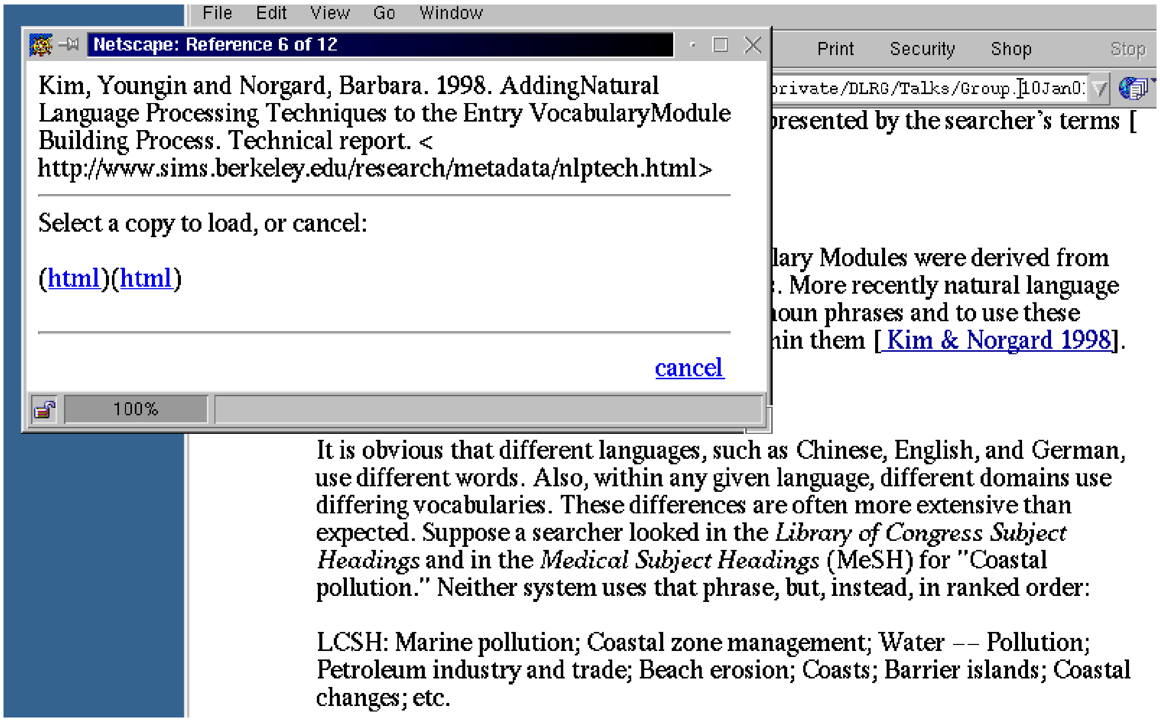

Figure 7: A Simple Reference Linking Application: clicking on the [`Kim & Norgard 1998`] reference brings up a selection of this work's on-line items.

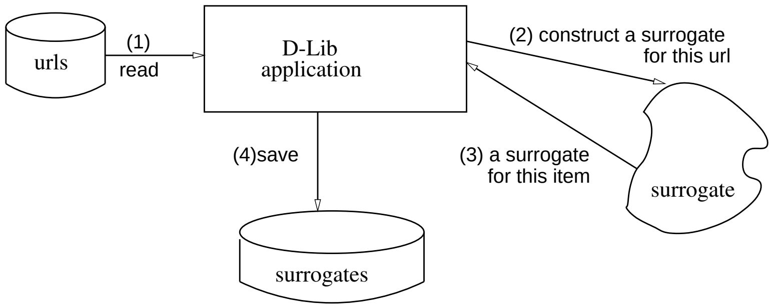

Figure 8: The application to intralink D-Lib.

Figure 8 illustrates the major steps in analyzing a D-Lib paper. The application, running from the command line, (1) inputs a file of D-Lib URLs, automatically generated from table of contents pages. For each URL, the application (2) constructs a surrogate object, which proceeds to extract reference linking information, (3) gets a handle to the surrogate, and (4) stores the surrogate. The Java code to perform this processing is:

```
Surrogate s = new Surrogate ( url );
s.save();
```

This example has shown the API being used in a static batch mode. The application's result is a repository of surrogates, which represents, and adds value to, the D-Lib on-line journal.

5 Evaluation of the Implementation

Because our approach extracts reference linking and bibliographic data *automatically* from widely variable sources, it cannot be expected that the data will be 100% accurate. Indeed, *automatic librarianship* inevitably means some tradeoff in functionality and accuracy in exchange for its cost savings[1]. Examine, for example, the Web search engines that are immensely useful despite their "imperfect" behavior. We believe that the accuracy of our approach, 80% in our experimentation, makes it a plausible basis for the creation of useful reference linking services.

There are two categories of parsing errors: incorrectly extracting bibliographic data about the item being analyzed; and incorrectly parsing the reference strings contained in the analyzed items. Both of these must be minimized if an efficient reference linking layer is to be attained. To measure our performance in this respect, we devised a performance metric based on both of these inputs.

For each item analyzed, the *item accuracy* is the number of elements parsed correctly, divided by the total number of elements in the item. Specifically, the elements used are: the item's title, the item's authors (each author counts as one element), the item's year of publication, the reference contexts (each context counts as one element) and the average reference accuracy times the number of reference strings.

The *reference accuracy* for one reference string is the per centage of correctly parsed elements, including title, authors, year, contexts, and URL (if present). The *average reference accuracy* is the average over all the references. Figure 9 shows a hypothetical item with its hypothetical references, hypothetical results, and calculation of the item accuracy, in this case 75%.

Reference Accuracy (16 reference strings)

Ordinal	Number Elements	Number correct	Reference Accuracy	Ordinal	Number Elements	Number correct	Reference Accuracy
1	7	4	57	9	4	3	75
2	5	1	20	10	5	5	100
3	5	5	100	11	5	5	100
4	5	5	100	12	8	6	75
5	7	7	100	13	5	2	40
6	5	5	100	14	6	6	100
7	4	1	25	15	5	1	20
8	7	6	86	16	4	1	25

Total Reference Accuracy = 1123; Average = 1123/16 = 70.19

Item Accuracy

What	How Many	How Many Correct	%
title	1	1	
authors	2	0	
year	1	1	
contexts	8	8	
references	16	11	
Totals	28	21	75%

Figure 9: Example of Item Accuracy for hypothetical item with 2 authors, 16 references and 8 reference contexts. First calculate the average Reference Accuracy (top figure, 70%). Then in the bottom table, use 70% of 16 (11) references as the average accuracy of reference parsing. The Item Accuracy metric is then 21 divided by 28, or 75%.

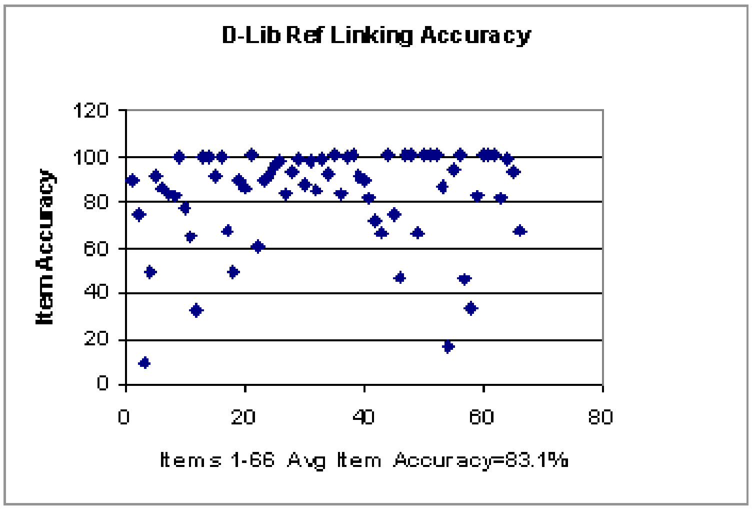

Figure 10: Item accuracies for a set of 66 D-Lib papers

For evaluation purposes, we selected a random set of 70 D-Lib papers. Of this number, 4 were not able to be converted to XML (i.e. XHTML) and so were discarded.[2] For the remainder, item accuracies were determined by inspection of the data contained in stored surrogates; the item accuracy for this group of 66 papers is plotted in Figure 10. Most accuracy points lie above 80%.

As explained in Figure 9, reference parsing accuracy is part of item accuracy. The first 66 items contained 504 references. Reference strings occur in a wide variety of formats; however, the reference strings were parsed accurately (see Figure 11). The overall level of accuracy is above 80%.

Some of the items in Figure 10 were very low for reasons such as the references being in a separate document, or the reference section of the paper not labeled as such, resulting in a zero "average reference parsing accuracy".

While the overall averages are acceptable, it is harder to concentrate accuracy into one place – that is, to parse all one item's metadata and each its references correctly. We therefore look at how often it was possible to perfectly extract a paper's metadata, which would correspond to the number of times the user would get a perfect answer in response to the `getMyData()` method. We also looked at how often reference strings in a paper are perfectly parsed, which corresponds to the quality of the response to the `getReferenceList` request. The results are contained in Tables 1 and 2. It appears that extracting

[2] We use JTidy, a tool from W3C, to convert HTML items to their XML form, XHTML. JTidy finds some HTML documents insusceptible to conversion, due to ambiguities in the HTML. See [3] for examples of what can go wrong.

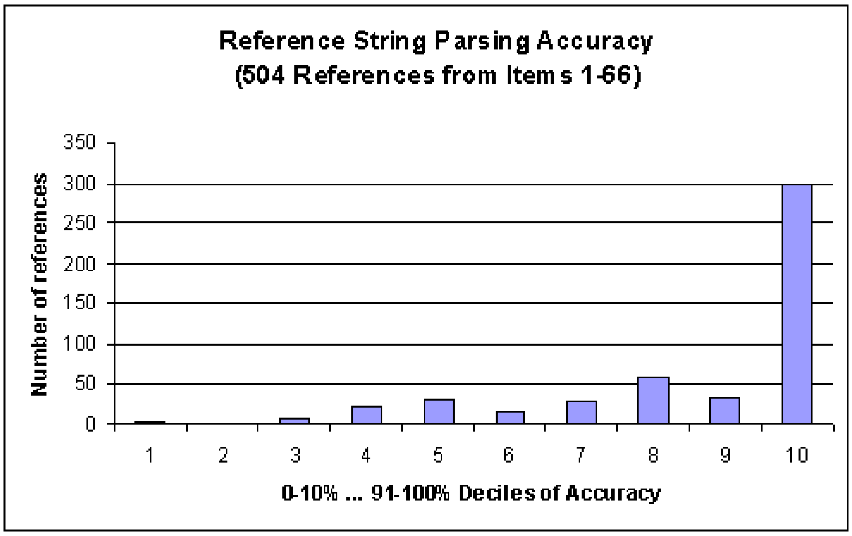

Figure 11: Parsing accuracy for the 504 reference strings contained in a set of 66 D-Lib papers. More than half are in the 91–100% range.

D-Lib 1995 to August 2000 Subsample: Metadata Extraction

Description	Number	% of Total
Number of D-Lib papers:	70	100
Converted to XHTML:	66	94
Extraction is Perfect	45	68
Good (70% or more)	9	14
Poor (below 70%)	12	18

Table 1: Number of D-Lib items whose bibliographic data was correctly extracted. The rightmost 2 columns are the 70-item subset of D-Lib papers. The bottom 3 rows are a per centage of row 2, that is, of the items that could be turned into XHTML. "Perfect extraction" means that all bibliographic data was accurately extracted.

D-Lib 1995 though August 2000 Sample

Description	Number	% of Total
Number of References:	504	100
Parsing is Perfect	288	57
Good (70% or more)	102	20
Poor (below 70%)	114	23

Table 2: Number of correctly parsed references in the 70-item sample of D-Lib Papers. The two right-most columns show the results for 66 D-Lib papers that contain 504 references

information from reference strings is more difficult than getting data about the item itself (57% vs. 68% perfection). However, in both cases, the majority of extraction is perfect.

6 Conclusions and Further Research

Automatic extraction of reference linking information is difficult, but possible. It is good enough to allow a reference linking layer to be added to on-line technical literature. The flexible object-oriented API makes it exceptionally easy to build new reference linking applications and tools, and we are doing some of that now. We are also extending our automatic extraction to PDF files, using Etymon's PJ class and tools from Adobe.

One area of interest, applicable to any automatic procedure, is how and how much to enhance machine processing with human effort. Using the API in batch mode, one could run an off-line "upgrade" procedure allowing human editing of surrogate encapsulated data. When an edited surrogate is resurrected, it will have the corrected information. The costs and benefits of this editing is one area of interesting research.

Another research area is data consistency. As more surrogates are instantiated, the same work could be encountered more than once (for example, as references in two different items). Slightly different data could exist for each instance. The problem then is to let the two surrogates pool their information so that both surrogates have a consistent world view. This requires either that the surrogates be able to find and communicate with each other, or to use a central database. Since a central database is to be avoided, it would be preferable to have the surrogates communicate among themselves. This is for future research.

The extraction of reference linking data is difficult mainly because parsing text produced by many different authors with many different conventions is problematical. However, we have found that there are a relatively limited set of variations in format, and have developed grammars to handle most of them. A separate paper [3] discusses this problem in more detail, and presents the algorithms we use.

At this point we are analyzing papers, examining the errors, patching up the Java API, and then analyzing new papers. As each additional paper gets processed, the implementation improves a little. At the present time, our reference linking API returns results that are 80–85% accurate, indicating that its architecture and design are sound.

References

[1] W. Arms. Automated digital libraries: How effectively can computers be used for the skill tasks of professional librarianship. *D-Lib Magazine*, July 2000.

[2] H. Atkins, C. Lyons, H. Ratner, C. Risher, C. Shillum, D. Sidman, and A. Stevens. Reference linking with DOIs: A case study. *D-Lib Magazine*, 6(2), February 2000. <http://www.dlib.org/dlib/february00/02risher.html>

[3] D. Bergmark. Automatic extraction of reference linking information from online documents. Technical Report TR 2000-1821, Cornell Computer Science Department, November 2000. in preparation.

[4] D. Bergmark and S. Keshav. Building blocks for IP telephony. *IEEE Communications Magazine*, 38(4):88–94, April 2000.

[5] Chaomei Chen and Les Carr. Trailablazing the literature of hypertext: Author co-citation analysis (1989–1998). In *ACM Hypertext '99*, Darmstadt, Germany, February 21-25 1999.

[6] E. Garfield. SCIENCE CITATION INDEX - a new dimension in indexing. *Science*, 144(3619):649, 1964.

[7] Monika R. Henzinger. Hyperlink analysis for the web. *IEEE Internet Computing*, 5(1):45–50, January/February 2001.

[8] Steve Hitchcock, Les Carr, Wendy Hall, Stephen Harris, S. Probets, D. Evans, and D. Brailsford. Linking electronic journals: Lessons from the Open Journal project. *D-Lib Magazine*, December 1998.

[9] Steve Hitchcock, Les Carr, Zhuoan Jiao, Donna Bergmark, Wendy Hall, Carl Lagoze, and Stevan Harnad. Developing services for open eprint archives: Globalisation, integration and the impact of links. In *ACM Proceedings of Digital Libraries, 2000 (DL2000)*, San Antonio, Texas, 2000.

[10] Charlotte Jenkins and Dave Inman. Server-side automatic metadata generation using Qualified Dublin Core and RDF. In *IEEE Proc. Kyoto International Conference on Digital Libraries*, Kyoto, Japan, November 2000.

[11] Steve Lawrence and C. Lee Giles. Context and page analysis for improved web search. *IEEE Internet Computing*, 2(4):38–46, July/August 1998.

[12] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Digital libraries and autonomous citation indexing. *IEEE Computer*, 32(6):67–71, 1999. <http://www.researchindex.com>

[13] Birdie MacLennan. Presentation and access issues for electronic journals in a medium-sized academic institution. *The Journal of Electronic Publishing*, 5(1), September 1999. <http://www.press.umich.edu/jep/05-01/maclennan.h tml.

[14] Sandra Payette and Carl Lagoze. Value-added surrogates for distributed content. *D-Lib Magazine*, 6(6), June 2000.

[15] K. G. Saur. Functional requirements for bibliographic records, 1998. UBCIM Publications - New Series Vol. 19.

[16] Elaine Svenonius. *The Intellectual Foundation of Information Organization*. M.I.T. Press, 2000.

[17] Herbert van de Sompel and Patrick Hochstenbach. Reference linking in a hybrid libary environment, part 2: SFX, a generic linking solution. *D-Lib Magazine*, 5(4), April 1999.