# Reference Linking the Web's Scholarly Papers

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. Thus the Web is becoming an efficient source of up-to-date information for the scientific researcher, and more and more 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 the information obtained from the Web is potentially 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 links 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* to enhance the reference-linking behavior of Web documents. Given the URL of an on-line paper, a surrogate object is constructed for that paper. The surrogate fetches the content of the document and parses it to automatically extract reference linking data. Applications can then use the surrogate to access this reference linking data, encoded in XML, via a well-defined API.

We use this API to reference link the D-Lib magazine, an on-line journal of technical papers relating to digital library research. Currently we are (automatically) extracting reference linking information from the papers in this journal with 80% accuracy.

**Keywords:** reference linking, OpCit, value-added surrogates
**Word Count (abstract):** 248 **(paper):** 4500 (estimated)

[*]DARPA/CNRI Grant #2057/57-02
[†]NSF Grant # IIS-9907892

## 1 Background and Motivation

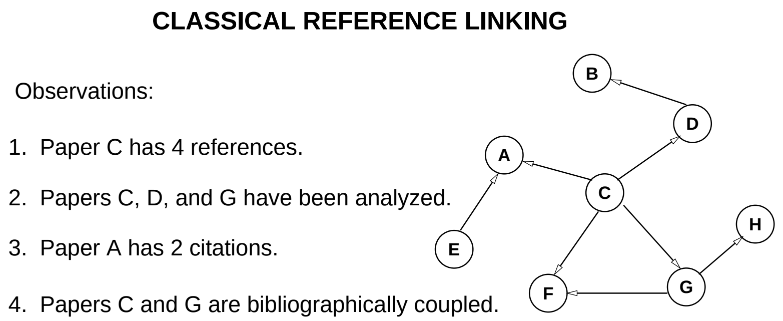
Along with the explosive growth of the Web has come a great increase in on-line scholarly literature. This literature comes in many forms. Informal on-line archives are repositories for papers and technical reports. Proceedings are more and more commonly published on the Web. The collection of on-line journals is growing. People sometimes just put their papers on their Web site. Thus the Web is becoming an efficient resource for up-to-date information for the scientific researcher, and more and more 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 the information obtained from the Web is potentially 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".

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[5] 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 metadata, 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, that Papers C, D, and G have been analyzed, that Paper A has two citations, and that Papers C and G are bibliographically coupled (i.e. they have a reference in common). The links in the graph are *explicitly* contained in the Science Citation Index.

We then fast-forward some 25 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. The unidirectionality of links on the Web makes the corresponding functionality, providing access to a paper's citations a more daunting task.

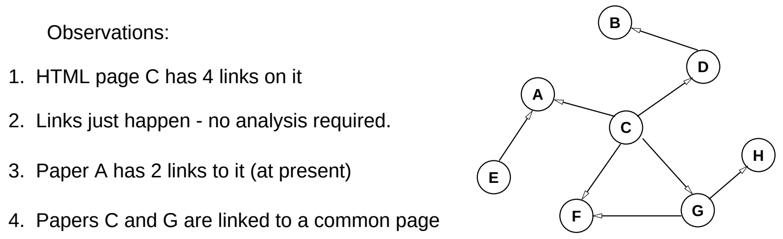


CLASSICAL REFERENCE LINKING

Observations:

1. Paper C has 4 references.
2. Papers C, D, and G have been analyzed.
3. Paper A has 2 citations.
4. Papers C and G are bibliographically coupled.

Figure 1: Classical Reference Linking



Observations:

1. HTML page C has 4 links on it
2. Links just happen - no analysis required.
3. Paper A has 2 links to it (at present)
4. Papers C and G are linked to a common page

Figure 2: Linking on the Web

Figure 2 shows just a portion of interlinked Web papers. From the fragment shown here, we can deduce the HTML page C has four links in it to other HTML pages; page A has at present two links to it; and papers C and G are linked to a common page. The graph is implicit; authors often fail to insert explicit links to references and citations can be discovered only by exhaustive analysis of the Web.

In our reference linking project we are aiming somewhere between the classical view and what exists today on the Web. We wish to make the graph in Figure 2 explicit, as well as automatically supply additional links where possible.

Our work aims to create 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. Our goal is to provide this layer *automatically*, without the direct participation from authors.

Figure 3 is just one 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]" will be a clickable live link, so that the user could start

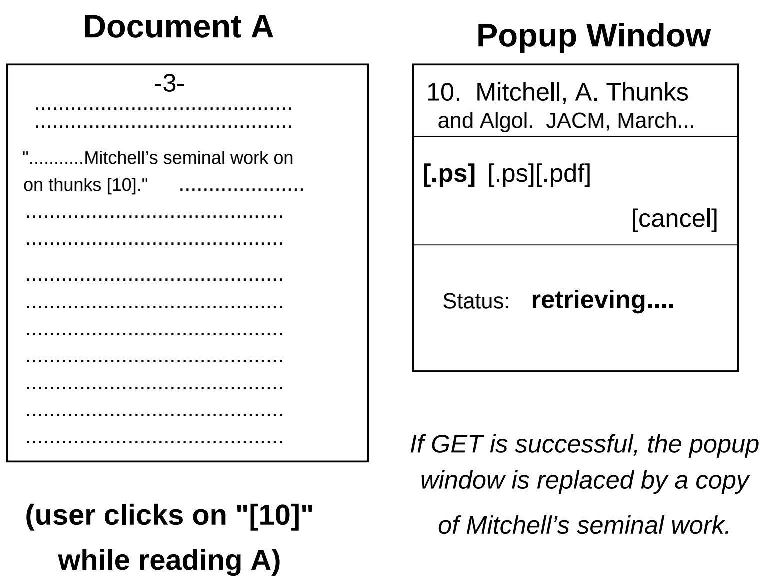

Document A
-3-
"...........Mitchell's seminal work on
on thunks [10]."
(user clicks on "[10]"
while reading A)
Popup Window
10. Mitchell, A. Thunks
and Algol. JACM, March...
[.ps] [.ps][.pdf]
[cancel]
Status: retrieving....
If GET is successful, the popup
window is replaced by a copy
of Mitchell's seminal work.

Figure 3: A Reference Linking Application

fetching that copy while continuing to read the original paper. One interface that would support this goal might be a JavaScript popup window that looks something like the one on the right side of Figure 3; the complete reference string is shown 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 an SFX[10] 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. Figure out 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 paper's reference section.
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.
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 you can turn this into an anchor that can be clicked. For other formats some kind of auxiliary display is needed.

From a data-flow perspective this process can be describe as *analysis*, *data access*, and *presentation*. Our work has been concentrating on the first two. We describe in this paper methods and results on the extraction of reference linking data from on-line literature. We also describe the 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 who have come together forming a group called CrossRef, an effort that grew out of the DOIX project described by Atkins [2]. This organization, consisting of professional journal publishers, is interlinking their on-line journals. Each publisher generates its own metadata and shares it with the other publishers in the group.

The ResearchIndex[7] is a well-known reference linking application which seeks to build a large on-line database of citation information, in the field of computer science. Their software is available to our project, and we have borrowed some of their techniques.

Formally, our work is part of a larger project called OpCit based at Southampton University. Funded jointly by JISC in the UK and the NSF Digital Libraries Initiative in the US, this project is reference linking arXhiv, the technical report repository at Los Alamos. The project's home page is at `<http://opcit.eprints.org>`. To read more about the origins of this project, refer to [6] which discusses the precursor of Opcit, the Open Journals project.

Finally, there is related work going on at Cornell. SFX, an advanced reference linking system that takes the user's context into account, is being developed by Van de Sompel, now a member of the Digital Library Research Group at Cornell. We are actively discussing the merging of his work with the work reported here. They are in fact quite complementary.

## 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 referred to by A. There is a subtle, but important, difference between A and B. A is an *Item*, something that has a format type, 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 findable on-line. Thus we say that Works exist in the form of zero or more Items. Our definition of Item deviates slightly from standard library usage in that it is limited to on-line copies. See Svenonius [9] for a good philosophical discussion of what a work is, and how the work and item distinction has played a role in traditional library cataloguing theory.

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

## 2.2 References and Citations

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

- *linkable references* are copies of work 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 in the document, 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. Southampton's work, for example, is analyzing the arXiv repository. At Cornell, we are focussing on the more general problem of architecting a reference layer for the Web that works irregardless of repository boundaries. It should also be noted that in our project, when we analyze respositories for reference linking information, we do not keep copies of any items.

## 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, we have a linkable reference.

If we analyze several repositories, then we can *interlink* items in these repositories. The extreme limit of this work, as more and more items are analyzsed, is to interlink the Web, at least the scholarly side of it.

## 3 The Reference Linking API

The reference linking architecture developed at Cornell is unique in several respects. First, because we are aiming to link on-line literature without author intervention, we are taking an *automatic* approach to parsing document source in order to extract the item's metadata as well as the metadata of the item's references. This contrasts with CrossRef's approach. 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 may be 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 databases to store information about works and items and do a lot of "database crunching". For example, there would 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 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. Reference linking 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. The surrogate also embodies the reference linking behavior of on-line items.

This use of surrogates for reference linking is consistent with our overall architectural approach in digital library research at Cornell. We make use of "value-added surrogates"[8] as a vehicle for endowing digital objects with a wide variety of extensible behaviors (e.g. preservation, access management).

A third unusual aspect of reference linking at Cornell is that we do not put together one monolithic reference linking service. Rather, we provide an API on top of which a variety of such services can be built. Such an approach has been quite successful in other, unrelated endeavors (see, for example, [3]).

Having an API specifies the operational semantics of reference linking; it also allows us to cleanly separate the analysis phase of reference linking from the presentation phase. The advantage of creating an API is that no decision is made in advance of what the data should be used for.

## 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, 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 (we have their bits).

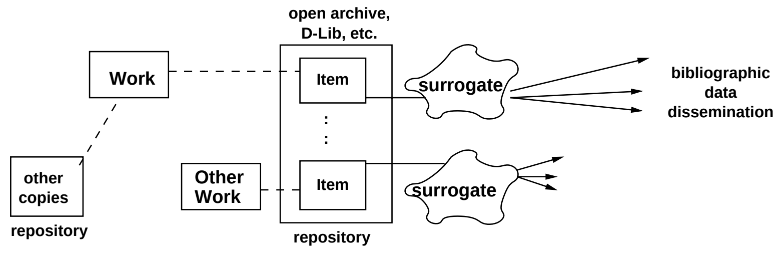


Figure 4: An Architecture for Reference Analysis

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

To the right of the archive items are the surrogates, shown as "blobs". They know how to disseminate bibliographic data about the item, and indirectly, about the work. As stated above, client applications ask the questions and then display or otherwise use the results. The API supplies the data.

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.

This paper does not cover issues related to storage, discovery of, and access to surrogates. In general, we envision the creation of specialized search services that will lead users and client applications to surrogates, rather than "raw" copies of scholarly papers, and thus make available the enhanced functionality of the surrogates.

## 3.2 Components of the API

The four defined methods in the Cornell reference linking API are:

- `getLinkedText` – returns the contents of the paper (as data) augmented with reference linking information. A primary 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 that wish to know what references are contained in this paper. For example, if one were building the SCI, this would be the question to ask, along with the next one.
- `getMyData` - this returns that paper's own metadata. It could have other uses; for example, one client might have a button labeled "get BibTeX"; when the button is pushed, the client invokes `getMyData` on the surrogate, and reformats 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.

## 3.3 Output from the API

Figure 4 showed the surrogates disseminating bibliographic information about their items, in response to a particular method in the API being invoked. Each method 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 XML data disseminated by `getReferenceList` contains one top level element, `<reference_list>`, which in turn consists of 0 or more `<reference>` elements.

Figure 5 shows the second reference (`ord="2"`) of an example surrogate's item. The `<reference>` element consists of:

1. The 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>`.

Note the "[2]" in the example's `<context>`. Since the reference string for the Maly paper contains a URL, this may become the anchor of a live link in any text returned by a call to this surrogate's `getLinkedText` method. Each reference anchor in the text returned by `getLinkedText` is enclosed in a `<reflink>` element, which is sufficiently rich to point to various on-line copies of the reference, to retrieve the reference string itself, and so on.

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

The XML data returned by each of the four methods in the API can then serve as input to other processing; it could be transformed by XSLT and rendered, or undergo further manipulation. This leads to the two-phased architecture shown in Figure 7 with structured data – XML – at the interface between the two phases.

## 3.4 A Java Implementation of the API

The API can be easily implemented in Java, Perl, or even as part of a larger protocol. Our Java implementation will briefly be discussed 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. Typically the item is translated to XML before further analysis. 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.

```
<? xml version="1.0" ?>
<api:reference_list length="17"
      xmlns:api="http://www.cs.cornell.edu/cdlrg/..."
      xmlns:dc="http://purl.org/DC">
  <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">[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.

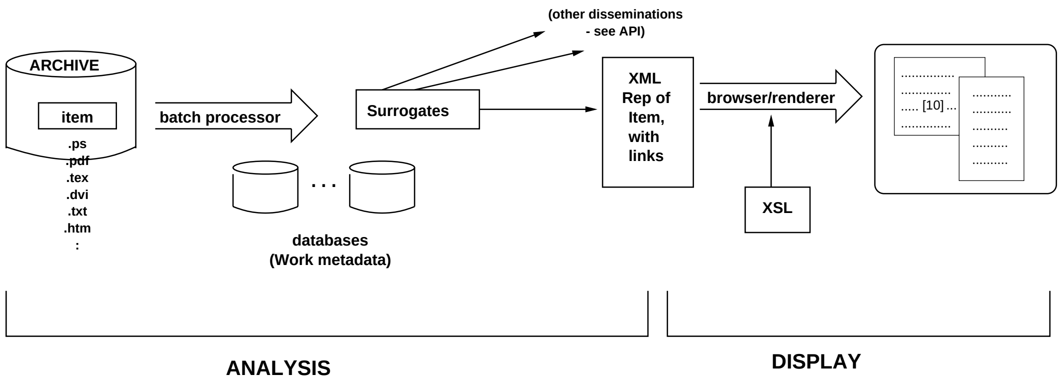


Figure 7: Overall Reference Linking Architecture

.

When the surrogate is returned to the client application, the item has been parsed and preliminary reference data has been stored into data fields within the surrogate. Invoking one of the four methods on the surrogate, e.g. `getLinkedText()`, causes further analysis of the reference data and culminates in an XML byte array.

The surrogates can be constructed and used on the fly and then discarded, or they may be stored for further use. This allows for a wide range of applications, from constructing a database of citation information to providing a completely dynamic reference linking service.

The Java implementation consists of less than 6000 lines of code and uses both DOM and SAX parsing of XML data.

## 4 Experiments with the Architecture

The reference linking API can be used for a large variety of applications. This section briefly sketches two of them.

### 4.1 A Simple Display Application

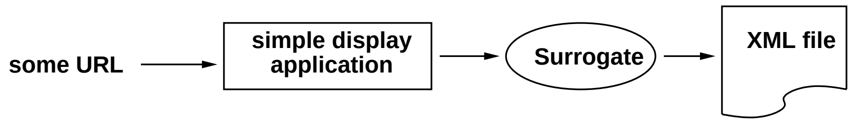


Figure 8: A Simple Reference Linking Application

The first example shows how on-demand surrogate creation can be used to provide a basis for a reference linked document viewer. A demo of this application is currently being built as part of a student project. In Figure 8, the client application is given the URL of some on-line item. The application 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.

The right-hand side of Figure 8 shows the client application invoking various methods on the 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 result of this request is a XML byte array, which is then passed to a routine, `display()`, which presents the linked text to a user. For a display similar to that shown in Figure 3, the steps in the presentation are as follows:

1. Run XSLT or a similar translator to convert the API's `<reflink>` elements into JavaScript code.
2. Display the translated XML object to the user.
3. When the user clicks on a reference anchor that has a live link, bring up the *retrieving...* dialogue, showing the complete reference string, and show what formats exist for this work.
4. If the user clicks on the cancel button, quit. Otherwise retrieve the format selected by the user and display it in a separate window.

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

### 4.2 Reference Linking the D-Lib Magazine

The second example is one which gathers and stores reference linking information for future use. We are currently using the Java implementation of the reference linking API to analyze D-Lib articles. D-Lib is an online 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 on the format of the papers submitted to the journal, and therefore provides a wide selection of paper layouts. All D-Lib articles are written in HTML.

Figure 9 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. (The file was automatically generated from D-Lib table of contents pages.) For each URL, the application (2) constructs a surrogate object, which proceeds to extract reference linking information, and (3) gets a handle to the surrogate; and (4) stores the surrogate. The Java code to perform this processing is as follows:

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

(The reference linking API contains, in addition to the four methods mentioned earlier, `save()` and `restore()`.

This example has shown the API being used in a static, or batch, mode.

## 5 Evaluation of the Implementation

Because our approach extracts all reference linking and bibliographic data *automatically*, it cannot be expected that the data will be 100%

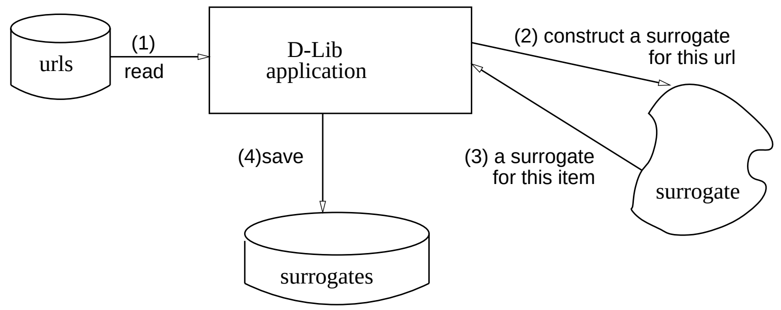


Figure 9: The application to intralink D-Lib.

accurate. Fortunately (unlike for library services) a reference linking service for on-line documents does not have to be completely accurate. Rather, one aims for the "sweet spot" where at least one copy of the reference can be retrieved (so *recall* is not that important), and where there are not too many false links (*precision* has to be good enough). We believe that an 80% accuracy level is the minimum acceptable threshhold that would permit interesting value added services.

Since our work with D-Lib resulted in a large number of surrogates, we simply examined their reference linking data for accuracy. Our current results show that we are very near to achieving our desired level of accuracy. The accuracy of our parsing improves as each new batch of papers is processed.

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. We therefore 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 its elements that are correctly parsed. These elements include: title, each author, year, contexts, and URL (if present). Figure 10 shows a hypothetical item with its hypothetical references, the hypothetical parsing results, and calcluation of the item accuracy, in this case 75%.

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. For the remainder, item accuracies were determined by human inspection of the data contained in stored surrogates; the item accuracy is plotted in Figure 11. As can be seen, most of the items lie above our desired 80% level of accuracy.

**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 10: 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%.

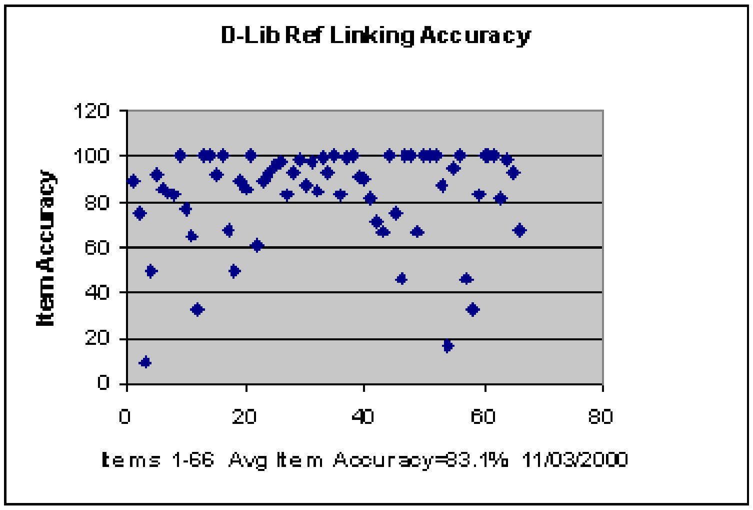


Figure 11: Item Accuracies for a set of 66 D-Lib papers

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.

For a given set of items, the number and kind of references in those items is much larger and varied than the set. Figures 12 and 13 show the accuracy of parsing the reference strings in the same set of D-Lib papers. Again the majority of the references parse to the desired degree of accuracy, with a surprising number parsed perfectly. The overall level of accuracy is above 80%.

While the overall averages are acceptable, it is harder to get accuracy concentrated into one place – that is, to parse all the 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

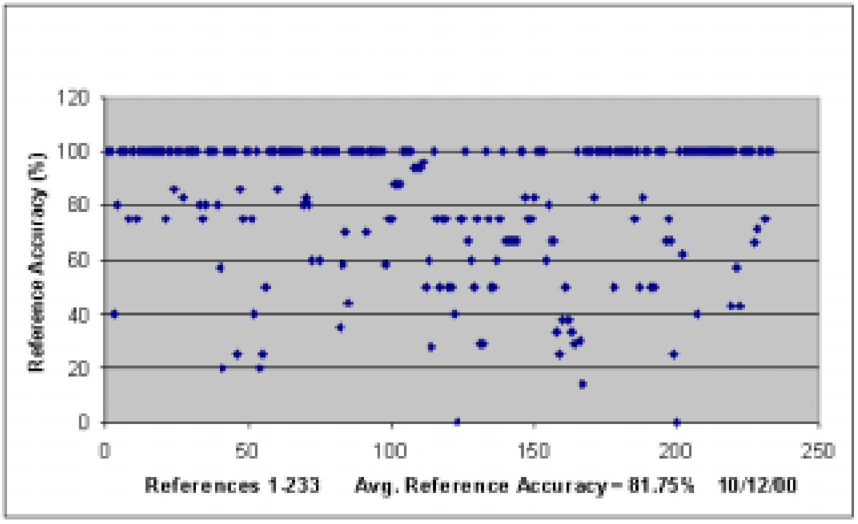


Figure 12: Reference Accuracies for a set of 66 D-Lib papers, batch 1.

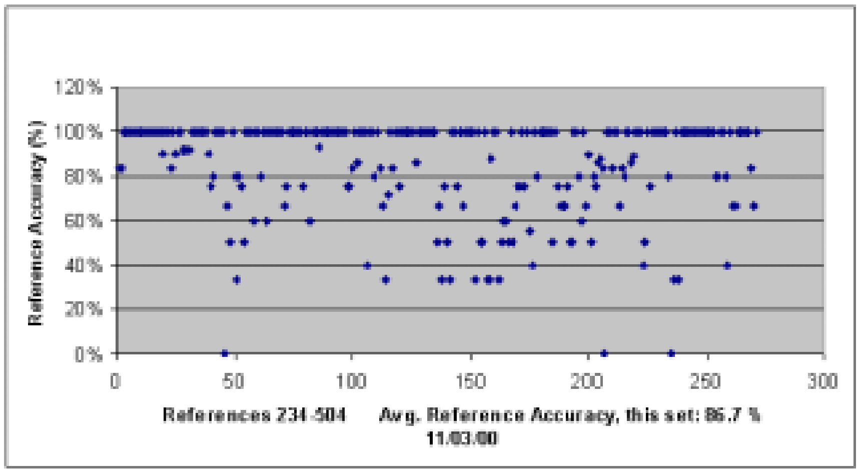


Figure 13: Reference Accuracies for a set of 66 D-Lib papers, batch 2.

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

corresponds to the quality of the response to the `getReferenceList` request. The results are contained in Tables 1 and 2.

## 6 Conclusions

This project shows that automatic extraction of reference linking information is difficult, but possible. The extraction of reference linking data is difficult mainly because parsing text produced by many different authors in many different formats with many different conventions is problematical. However, we have found that there are a relatively limited set of variations in format, and have successfully developed grammars to handle most of them. A separate paper [4] discusses this problem in more detail, and presents some algorithms for extracting reference linking information.

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. If we look at the *proportion* of elements that can be correctly extracted from an item or from a reference, we have more than 80% item and reference accuracy.

Of course, using any available metadata would improve this accuracy, since then we would need only to handle context extraction and matching reference strings to contexts. But in our D-Lib analysis, such metadata has only recently begun to be available, and so we extract this information ourselves, automatically.

ResearchIndex also automatically extracts data from items discovered online, and does a remarkably good job. Its main strength lies in applying clustering methods and other artificial intelligence techniques to the analyzed material. Our software does not incorporate AI methods, but does almost as well.

## 7 Further Research

When the API is used in batch mode (where surrogates are saved for re-use), it might be useful to run an offline "upgrade" procedure which allows human editing of surrogate encapsulated data. When an edited surrogate is resurrected, it will have the corrected information. How to expedite this process is one potential area for research.

Another research area is data consistency. Over the course of time many surrogates are instantiated and 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 that "know" about each version pool their information so that both surrogates have consistent data.

This requires either that the surrogates be able to find and communicate with each other, or that there be a central database that both surrogates could consult.

We might investigate in future research the problem of arranging for the surrogates to communicate among themselves. For now we keep a small database of works seen so far which at least allows sketchy information to be updated.

The work done so far indicates that the architecture and design for the reference linking API are sound. The flexible object-oriented API makes it exceptionally easy to build new reference linking applications.

## References

[1] W. Arms. Automated digital libraries: How effectviely 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. Refererence linking with DOIs: A case study. *D-Lib Magazine*, 6(2), February 2000. <http://www.dlib.org/dlib/february00/02risher.html>

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

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

[5] E. Garfield. Science Citation Index - a new dimension in indexing. *Science*, 144(3619):649, 1964.

[6] 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.

[7] 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>

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

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

[10] 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.