Hypertext by Link-Resolving Components

Frank Wm. Tompa
G. Elizabeth Blake
Darrell R. Raymond

Centre for the New OED and Text Research
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

In conventional hypertexts, links are explicit connections between specific regions of a text. We describe an architecture that treats links as the outcome of responses to user pokes. In this architecture, a hypertext is a collection of link-resolving components, each interpreting a user's request according to its own resolution algorithm. Each link-resolving component is a set of cooperative processes, communicating with a central network manager. When a user points at some location within a window, each link-resolving component is given a key derived according to a previously-stored algorithm; the link-resolving components concurrently update according to their algorithms for resolving the keys. Multiple applications can easily share a common source and be invoked simultaneously, providing a concurrent browsing mechanism. Two example hypertexts employing this architecture are described.

Keywords: hypertext system architecture, dynamic linking, link resolution, database keys, *Oxford English Dictionary*

1. Introduction.

The fundamental nature of hypertext links has changed little since their first appearance: a link is an explicit label, selected with a pointing device, and leading to a specific location in the text. Improved graphical capabilities have changed the look of these links—they can be iconic buttons, have sound attached, be embedded in the text, and so on—but their basic nature has remained largely unchanged. Several authors have argued that it is impractical to build large hypertexts with this technique [Bieber91, DeRose89, Halasz91, Raymond88]. In general, large texts have too many links of too many different types to be able to create them all manually, store them all explicitly, and display them all iconically. Even if one were able to do so, the density of interconnection would overwhelm most users. For highly dynamic texts, link maintenance is an added complication.

A second problem with explicit links results because they presuppose that the link's target is fixed, given its source. The user, on the other hand, when making a poke into the hypertext, wants the system to 'do the right thing'. What 'the right thing' is depends partly on the document, partly on the context surrounding the poke, partly on the interaction leading up to the poke, and partly on the user's intent. For example, when reading Shakespeare, a user poking at the word 'froward in the line

174 Come, come, you froward and unable worms,

may want to do one (or more, or all) of several things. He or she may wish to find other occurrences of that word within the same work; or possibly within any of Shakespeare's works; or possibly within another author's works; or possibly within a whole corpus of text. Alternatively, the user may wish to find the definition of the word in the *OED*, or in some other dictionary; or the user may wish to examine that passage only in variant editions, or to read authoritative commentary on the word or the line; and so forth. In fact, there are unboundedly many possible interpretations of the user's 'poke at the screen.

The location of a poke does not uniquely determine a link's meaning. However, a potential target application can often decide the meaning of a poke *as it applies to the application*. A poke at a word in Shakespeare might mean anything, but from the point of view of the dictionary-lookup application, it almost certainly means to look up the word in the dictionary. Similarly, the meaning of a poke at a Shakespearean quotation in the *Oxford English Dictionary* is determined by whether the target is a bibliographic lookup tool (in which case, the poke means to display full bibliographic information about the quotation) or the Shakespeare lookup tool (in which case, the poke requests the quote in the context of the full play). We refer to applications that can narrow the meaning of a poke as *link-resolving components*.

The notion of link-resolving components changes the notion of hypertext and its links. Instead of defining a hypertext as a document with interconnecting links, we define it as a collection of link-resolving components. Links then become similar to database keys: not only can individual hypertext links be stored as explicit pointers, as in most other systems, but they can also be resolved 'on the fly through the execution of more complicated algorithms. We next describe an architecture based on link-resolving components.

2. Architecture.

A hypertext based on link-resolving components is almost inherently distributed, unlike conventional hypertexts, which are often monolithic. Each component in a link-resolving system is an independent entity, though not necessarily with an independent text. Instead, the lines of demarcation are drawn according to the link-resolving algorithm. For example, one may have several link-resolving components that operate over the Shakespeare corpus; given a poke, the first might find all matches within a given play, a second might find matches in other plays, a third might find matches to dialogue of the given character, and so on.

Users are likely to want to combine several hypertexts in one environment; for example, a hypertext on Shakespeare's works should be able to co-exist with one on the *OED*. Links may cross from Shakespeare to the *OED* (to look up words), and from the *OED* to Shakespeare (to find the contexts of citations). Thus, in general, link selection operates on one sub-text and link resolution operates on another.

Since the components isolate their texts from the rest of the system, a link-resolving component cannot expect to access the text to determine the source of the link. If they are to be able to operate independently, neither the hypertext application containing the source of the link nor the one containing the target can be fully responsible for link traversal. Thus, each link-resolving component must inform the other components of what context it needs to resolve a link. In effect, it must define what it will consider as a *key*. Link-resolving components must be able to specify their key in a general way, so that other components can return them. For example, the dictionary lookup component requires a word as a key, and it must specify what constitutes a 'word', for its purposes. A play lookup component, on the other hand, might need to have the position of the containing line (i.e., Act V, Scene 2, line 174) of the poke, so that it can resolve the link (in the case above, it would be resolved to *The Taming of the Shrew*).

Our architecture realizes a link-resolving design by using concurrent components communicating via a central handler. Each link-resolving component registers with the central handler a description of the keys that it can accept; when a user pokes at a window, the component controlling that window is responsible for converting that poke into a key that can be used by the targets. For example, one component may specify that its keys are blank-delimited strings, whereas another may specify that its keys are pointers to segments of text beginning with the string ‘<Q>. The central handler distributes this information to each cooperating component. Subsequently, if a component receives a user’s poke, it converts that action to the type of key requested by each active link-resolving component, and sends the set of keys back to the handler. The central handler then distributes the keys to the respective link-resolution components, each of which resolves its key independently.

Since it manages the key descriptions and the communication of keys, the handler is a convenient place to control the interaction of the link-resolving components. In most hypertexts, not all of the pokes should be interpreted by every link-resolving component. When an integrated hypertext application is created, the designer must specify the link-resolution components, and may specify some restrictions in the distribution of keys generated by those components. This setup specification is further limited by a runtime switching facility, which determines the routing of keys for individual pokes.

3. Prototype implementation.

A prototype system has been implemented to study this architecture in practical use. In the prototype, components consist of sets of cooperating concurrent processes running under the UNIX operating system. Text database searches are typically performed by PAT [Salminen92, OpenText93a], and text display is provided mainly by LECTOR [Raymond92, OpenText93b] which runs on the X Window System.† Communication among processes is by tagged text, to enhance data and machine independence. The tagged text is parsed and manipulated using the *gdl* subroutine library, an implementation of grammar-defined text operators [Gonnet87].

The handler, as described above, is the central component of the prototype. The handler sets up the network of link-resolving processes, it asynchronously receives potential keys from any process, and passes properly formatted keys to each active process for resolution.

Associated with the handler is a window that displays the current network configuration. In our prototype this display is simply a list of the link-resolving components. A marker beside each component indicates whether it is currently active—that is, whether it will attempt to resolve key generated by some other component. Inactive components will not respond to any key. Each component can be toggled between active and inactive by poking at this display; that is, the display also acts as a menu.

When the handler is initiated, it configures the network according to an input specification file. The specification file contains tagged text that describes each component, the type of key that it needs, and the connectivity of the network. An example is shown in Figure 1. Nodes in the network are labeled according to the content of the *<Name>* field. Each node is associated with a link-resolving component specified by the process in the *<Exec>* field, with parameters

† UNIX is a trademark of AT&T, PAT and LECTOR are trademarks of Open Text Corporation, and X Windows is a trademark of the Massachusetts Institute of Technology.

```
<Proc>
     <Name>def</Name>
     <Exec>demo/run.def</Exec>
     <Arg>demo/def.parameter</Arg>
     <Menu>Lookup Definitions</Menu>
     <Active>yes</Active>
     <Open>yes</Open>
     <FindKey><Want>String</Want>
          <Method>
              <Delim>
               <StStr> </StStr>
               <StStr>></StStr>
               <EndStr> </EndStr>
               <EndStr>.</EndStr>
               <EndStr>,</EndStr>
              </Delim>
          </Method>
     </FindKey>
</Proc>
<Proc>
     <Name>biblio</Name>
     <Exec>demo/biblio.lookup</Exec>
     <Menu>Find Bibliographic Match</Menu>
     <Active>no</Active>
     <Open>yes</Open>
     <FindKey>
          <Want>String</Want>
          <Method><Docs>Q</Docs></Method>
     </FindKey>
</Proc>
<Proc>
     <Name>xref</Name>
     <Exec>demo/find.refs</Exec>
     <Menu>Follow Cross References<Menu>
     <Active>yes</Active>
     <Open>yes</Open>
     <FindKey>
          <Want>Region</Want>
          <Method><Docs>XR</Docs></Method>
     </FindKey>
</Proc>
<NoConn>
     <Src>biblio</Src>
     <Dest>xref</Dest>
     <Dest>biblio</Dest>
</NoConn>
```

Figure 1. Specification for a hypertext network configuration.

specified by *<Arg>* (the component itself may be a set of cooperating processes, but the one indicated by *<Exec>* is the one which communicates with this particular hypertext). The specification in Figure 1 declares three nodes: `def` corresponds to program `demo/run.def` with argument `demo/def.parameter`; `biblio` corresponds to program `demo/biblio.lookup`; and `xref` corresponds to program `demo/find.refs`. It is possible to execute several processes running the identical program,(perhaps with different arguments); the node labels serve as unique process identifiers. The *<Menu>* field contains a descriptive string that appears on the network configuration menu. For the example in Figure 1, this menu will comprise

× Lookup Definitions
× Find Bibliographic Match
× Follow Cross References

For each link-resolving component, the specification file includes a declaration stating whether the process is to be initially active (that is, accepting links to be resolved) and whether its display window is to be initially open (‘active implies ‘open).

Specification of the key for a link-resolving component includes declaring the type of key and the specific method by which it is to be determined. In our prototype, two types of key are supported: a string of text, and a text region (stored as offsets into a file). If a string is requested, *<Method>* indicates the delimiters of this string. If a region is requested, *<Method>* must indicate which region. In Figure 1, `def` declares a string as key, where the string is the shortest sequence of characters surrounding the poked location, starting with a blank or right angle bracket, and ending with a blank, comma, or period; `biblio` similarly expects a string, but its extent is determined by the PAT region type [Salminen92] named ‘Q; and `xref` expects a pair of pointers indicating the extent of the PAT region of type ‘XR surrounding the text position picked by the user.

Finally, the specification file may also contain one or more declarations that restrict communication between specific components. By default, every node in the network is connected to every other node; that is, every link-resolving component gets a key for every poke. To limit the network so that component *A* will never receive a key from component *B*, the file includes a *<NoConn>* declaration. In the example of Figure 1, keys from `biblio` cannot be sent to `xref`, nor to `biblio` itself.

A link-resolving component handles keys that result from pokes in other components, and it may accept pokes that it will convert to keys for other components. It communicates these keys to the handler, which forwards them to the other active link-resolving components. Generally, a link-resolving component manages a window in which text is displayed; it will update this display based on keys it receives, and interpret the display when pokes are made to it.

Each link-resolving component receives, on startup, information from the handler that identifies the other link-resolving components, their desired keys, and a method to convert a text offset to a key. For example, Figure 2 shows the startup information sent to a process declaring keys for the components declared in Figure 1. Notice that the *<FindKey>* field is taken directly from the network specification file.

Given the above network, a user pokes at a position on the screen by clicking the mouse button within some text display window. This poke is then converted to a data element by the component. LECTOR, for example, converts each poke to the text offset in the corresponding data file [Raymond92]. Using the information received as key specifications, the component must then find a key for each of the connected, active processes by extracting surrounding text,

```
<DataId>KeyInfo</DataId>
<KeyInfo>
   <Proc><ProcId>0</ProcId>
         <FindKey><Want>String</Want>
                  <Method><Delim><StStr> </StStr>
                                 <StStr>></StStr>
                                 <EndStr> </EndStr>
                                 <EndStr>.</EndStr>
                                 <EndStr>,</EndStr>
                          </Delim></Method>
         </FindKey></Proc>
   <Proc><ProcId>1</ProcId>
         <FindKey><Want>String</Want>
                  <Method><Docs>Q</Docs></Method>
         </FindKey></Proc>
   <Proc><ProcId>2</ProcId>
         <FindKey><Want>Region</Want>
                  <Method><Docs>XR</Docs></Method>
         </FindKey></Proc>
</KeyInfo>
```

Figure 2. Key specification.

calculating offsets of a surrounding region, and so forth, as indicated by each *<FindKey>* field. Failure to find a key for some component is not considered an error; it is assumed in this case that the poke does not correspond to the source of a link for that component.

The extracted keys are sent to the handler via tagged text as shown in Figure 3. The handler distributes the keys to the appropriate link-resolving components.

```
<LinkData>
   <Proc><ProcId>0</ProcId>
         <Link><String>manyfold</String></Link></Proc>
   <Proc><ProcId>2</ProcId>
         <Link><Region><Start>296285297</Start>
                      <End>296294583</End></Region>
               <File>/dicts/oed2e</File></Link></Proc>
</LinkData>
```

Figure 3. Extracted keys sent to the handler.

A user's poke finally results in each link-resolving component receiving a key that it can resolve, as shown in Figure 4.

A component that receives such a key can process it in any manner it thinks reasonable. The simplest thing to do is display the text of the key itself (this is useful for debugging). More commonly, the component looks up the key in a predefined table, and displays the corresponding information. This is essentially the same as following a stored pointer, the conventional method for handling hypertext links. A link-resolving component can also manipulate the key, perhaps by altering the corresponding text, and then perform database searches with the result, displaying

```
<DataId>Link</DataId>
<Link><Region><Start>296285297</Start>
              <End>296294583</End></Region>
      <File>/dicts/oed2e</File></Link>
```

Figure 4. Key sent to a link-resolving component.

the results in its window. Thus the architecture provides a range of capabilities from the conventional pre-computed link, to highly dynamic, content-based link resolution.

In our prototype, a user's poke in one window invokes each component to determine whether a link for that component has been selected for traversal. Typically, if a component receives key information that is incompatible with its resolution routines, it ignores the key—the poke might generate a false key if it is not a link source for this component.

We next present two applications built using this prototype.

4. Example 1: Literary scholar's application.

The first example is a hypertext interface for several scholarly documents, including the *Oxford English Dictionary*, the Bible, and the Shakespeare corpus. The hypertext contains link-resolution components that perform a number of functions useful for looking up words in these tests. Most of the link-resolving components themselves comprise two or more processes, one of which serves as display manager. The process structure of this application is shown in Figure 5; the user's view of the application is shown in Figure 6.

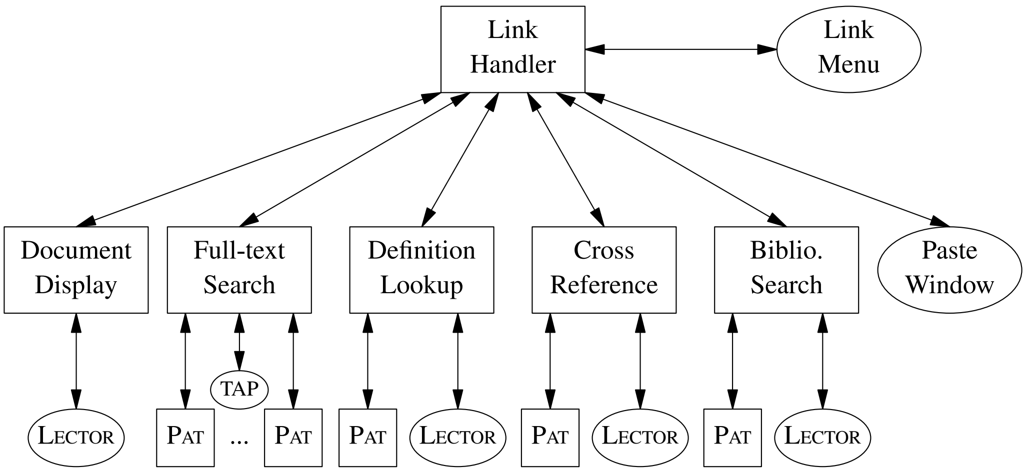

Figure 5. Process description of the *OED* hypertext utility

As can be seen in Figure 5, several displays are implemented with Lector; poking on any character in a Lector window selects the corresponding text offset, which is used to find the key.

Document Display. The simplest component expects a key containing an offset and a file name. It extracts from the file the sub-document containing this offset and displays it in a Lector window. This application can be used to select and display an entry from the *OED*, a chapter in the Bible, or one of Shakespeare's works. In Figure 6, the entry for 'dictionary is displayed.

Figure 6. Screen display for the *OED* hypertext utility.

Full-Text Search. This component allows the user to perform text searches on the *OED*, the Bible, or the Shakespeare corpus. The component expects a key in the form of a string, which it uses to perform a full-text search of the current database. The results of these searches are displayed in the corresponding TAP window as a list of matched lines of text within context. In Figure 6, a subset of lines matched by a search for the word ‘dictionary is displayed. Complex searches can be invoked by entering one or more PAT search commands [Salminen92] directly into this window. Poking at the window is interpreted as a request to follow the links defined at the corresponding match point (i.e., the start of the string matched by the full text search, which resulted in that line being displayed). In Figure 6, one line was thus selected, which is interpreted as a poke at character offset 102582987 into the *OED* (the start of a particular occurrence of the word ‘dictionary.)

OED Definition Lookup. This component finds the definitions of specific words in the *OED*, using a string as its key. The specification file declares that components should convert a poke to a string by looking forward and backward from the corresponding text offset for specified delimiting characters, including a blank and certain punctuation. These keys are resolved by searching for the relevant entries in the *OED*. This is not completely trivial, since a word may be defined in isolation, or as part of a phrase or compound, or as a morphological variant (i.e., the singular form). The definition lookup module handles these cases and, if successful, returns all entries that define this word, displaying them in a LECTOR window. For example, in Figure 6, a poke at the word ‘language or ‘languages in some window (perhaps the *Document Display* window in this example) caused all matching entries (e.g., for ‘language itself, the phrase ‘assembler language, etc.) to be displayed—LECTOR provides facilities to view subsequently the associated definitions within this component’s window. If the number of entries found is very large (for example, *man* is defined in isolation or in compounds 117 times in the *OED*), the results are restricted to entries where the string is the headword, and a warning to this effect is given.

OED Cross Reference Resolution: This component resolves a cross reference within the *OED*. The component expects a key describing a region of text, that is, two offsets (for the start and end points) and a file name. According to the description in the specification file, such a region is to be found by identifying the PAT region of type ‘XR containing the user-selected offset. If the file name in the key is that of the *OED*, the link resolution locates the entry containing the target of this cross reference and displays it in a LECTOR window, highlighting the specific target [Raymond90]. The corresponding window in Figure 6 shows the entry for ‘diction which could have resulted from poking at the cross-reference in the etymology for ‘dictionary in the *Document Display* window.

The cross-reference component must deal with a large variety in cross-reference specification. Consider the following:

If a cross reference is ambiguous (as for FAULT, which could refer to either the entry describing its use as a verb or the one describing its use as a noun), an appropriate message appears in the display window, together with the possible targets for the cross reference.

OED Bibliographic Search: This component matches a citation from the *OED* to an entry in the bibliographic database of authors and their works. The link expects a key containing a string and a file name, where the string is extracted from the text by isolating the PAT region of type ‘Q containing the selected offset. For example, one might select a string such as:

1555 EDEN *Decades* 27 They harde the noyse of the flutes, shalmes, and drummes.

see FAULT	
see CA'ING-WHALE	
see PER CAPUT	
see FLEET *v.*	verb entry
see -ATE^2	2nd suffix entry
see SOLSTICE 2	sense 2 within the entry
see UN-^{1} 8	1st prefix entry, sense 8
see STYLE *sb.* 8	noun entry, sense 8
prec.	the previous entry
4	sense 4 within the current entry

Figure 7. Example cross-references from the *OED*

Given this quotation string, the component will attempt to resolve it by matching it in the *OED*'s bibliographic database [Townsend89]. If a unique match is found, the bibliographic entry is displayed using LECTOR. If the match is ambiguous or no suitable target is found, a message to this effect is displayed instead. The corresponding window in Figure 6 shows the result of poking at the first citation in the entry for 'diction as displayed in the cross-reference window.

Paste: This component displays a window in which a string may be pasted using the middle button. It is not defined to be a link-resolution component, and so it does not accept a key from any other component. However, whenever a string is pasted into the window, it generates a key of type *string* (with no file), which is sent to all other active processes.

In this hypertext, not every type of key should be sent to every component. Thus, there are several restrictions limiting the communication network: *Full-Text Search* sends information only to the *Document Display* component, and does not receive information from *OED Definition Lookup*, *OED Cross Reference*, or *OED Bibliographic* components. The user can enable or disable any of the components by clicking in the *Link Menu* window.

Both the Bibliographic Search component and the Cross-Reference component were applications that existed before we began the work on link resolution [Townsend89, Raymond90]. These applications were easily integrated into a link-resolution architecture, whereas they would have been much more difficult to integrate into a monolithic stored-link architecture.

5. Example 2: Administrative data browser application.

The second example illustrates the use of the link-resolution architecture to browse administrative information, such as is found in a university or other business that requires documents describing policies and procedures, minutes of various bodies and committees, and calendars and schedules. Figures 8 and 9 show the process structure and screen layout for this application. The *Full-Text Search* and *Paste* components are identical in behaviour to those in the *OED* hypertext.

Print Confirmation. This component writes a document to a file in anticipation of printing it. Because administrative data is often created using several dissimilar word processing formats and systems, documents are converted to a tagged form prior to searching. The original form, however, is typically more suitable for printing. This component expects a key in the form of an offset and a file name. As with the *Document Display* component in the *OED* hypertext, the offset is converted to a region of text, but because the search component uses a file containing modified text, the offset must first be re-converted to a position in the original text file. This is done

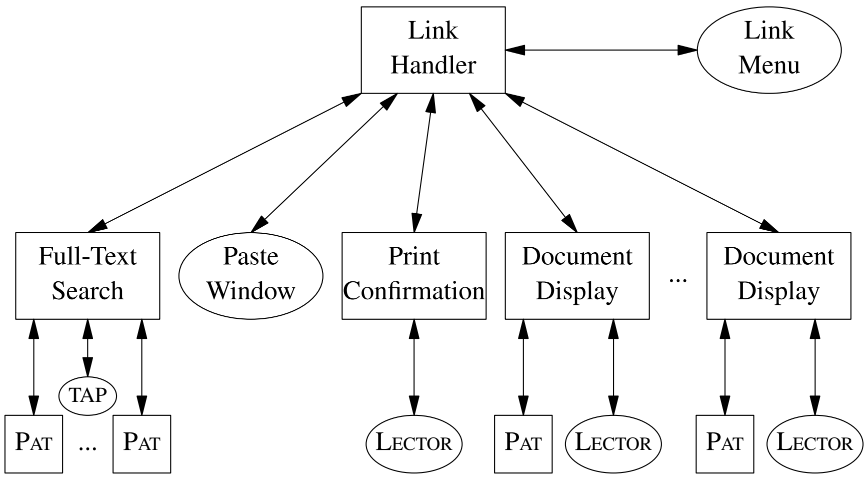

Figure 9: Process description of the administrative data browser

through a table lookup in a table constructed when the text conversion was performed. The component then notes in its window that it will copy text to some specified file, and requests verification from the user before proceeding. In this case of link resolution, not only is a key generated, and a database search conducted; print action routines are also invoked. Note that the display window is not suitable for link selection.

Document Display. Each document class has a corresponding component for displaying selected documents of that class in a LECTOR window. For example, each of the University databases (policies and procedures, minutes, and course calendar) have a component of this form. Each such component accepts two different types of key, and reacts depending on which type is received. If the key is an offset and file name (matching this component's data file), then the corresponding sub-document containing this offset is displayed (as is done by the *Document Display* component in the *OED* utility). On the other hand, given a key in the form of a string, the database is searched to find matches to the string. Thereafter, sub-documents containing the string are ranked according to the number of matches they contain, and the top-ranked documents are selected for display (the number of documents to select is determined by a parameter to the program). Within these component, the user may thereafter request the next ranked set of documents to be displayed.

6. Related systems and further work.

There are three important classes of systems related to the link-resolution approach. The first class is that of *link servers* [Pearl89]. A link server maintains a central database of links that can be accessed by hypertext applications. Like the link-resolution approach, this technique aims to provide a more open environment. While a link-resolution handler merely dispatches (and perhaps stores) descriptions of how keys are determined, a link server actually stores links. In database terms, a link server stores instances of a schema, while a handler only manages the schema; the instances remain under the control of the individual components. Since it manages an instance, a link server must deal with the problems of update; if a document is modified or a new version is requested, the link server must be updated to reflect all link changes. This introduces issues of consistency (how is one to deal with existing links that suddenly have no target?)

Figure 9. Screen display for the administrative data browser.

and synchronization (what does one do with users who are following links that no longer exist?). Such issues are not confronted by the link-resolution handler, since the target of a link is determined by a component, and not by data stored in a central server. If a document changes, its component will interpret all keys in light of the new document.

The second class of related work is system integration software. System integration software is essentially a flexible scheme for defining communications protocols to be used by distributed applications. The intent is to provide robust, machine-independent message-passing capabilities, along with associated semantics. There are several system integration packages, including Microsoft's OLE, Apple's AppleEvents, and of more recent prominence, Sun's ToolTalk and Hewlett-Packard's BMS [Jacobs93, Julienne93, Fromme93]. Processes using such a protocol can *notify* other processes of some event, or *request* that some activity be taken (usually with a return message). To some extent, our prototype software has defined a crude system integration protocol. There are doubtless advantages to using one of the more polished products (although choosing one now is difficult, since the technology is still evolving). System integration software by itself, however, does not define the link-resolution protocol nor the switching mechanism, which are the two main contributions of our work.

The third area of related work is in distributed relational databases. Our hypertexts are distributed databases to the extent that they involve data separated by logical (and possibly physical) interfaces. Relational databases, however, can make use of their schemas and the relational algebra as their means of interconnection; all the possible connections are already defined and accessible through powerful abstractions. Hypertext documents, on the other hand, generally lack both a schema *and* an algebra. Link-resolution makes a start on addressing this issue, since in effect a link-resolution protocol is a way of defining and using a foreign key.

There are several areas for further work. We need to expand the set of link resolution algorithms to handle more of the common cases. For example, accepting a key in the form of a string such as ‘refid=1234 and causing a search for an SGML element with attribute ‘id=1234 should not require the hypertext designer to write any code; instead, a descriptive specification should be supported. On the other hand, if link resolution requires full Turing computability, it may be very difficult to demonstrate the correctness of the algorithm. Ideally, link resolution algorithms would be written in a special, limited language that provides access to a set of link manipulation routines.

A second area is the need for better control over the connectivity of link-resolution components. Better display of the logical graph, for example, should be accompanied by the ability to do more than switch individual nodes off and on.

For this utility to be suitable in a commercial setting, conventional database capabilities would have to be added. For example, data security features are needed to limit access to certain types of data. Transaction control is needed to ensure that updates do not interfere with the normal use of the system. It would also be useful to add the ability to query traditional relational databases, and to accept the keys of these databases as keys for link resolution.

Acknowledgements

Tim Snider offered invaluable advice and assistance at all stages of the research. Financial support was provided by the University of Waterloo, by the Information Technology Research Centre and by the Natural Sciences and Engineering Research Council of Canada.

References

[Bieber91] M. Bieber and T. Isakowitz, ‘Valuation links: extending the computational power of hypertext, Tech. Rept. BCCS-91-1, Computer Science Department, Boston College (April 1991) 9 pp.

[DeRose89] S.J. DeRose, ‘Expanding the notion of links, *Proc. Hypertext 89*, Pittsburgh (November 1989) 249-258.

[Fromme93] B. Fromme, ‘Why You Need BMS, *SunExpert Magazine* (March 1993) 60-70.

[Gonnet87] G.H. Gonnet and F.W. Tompa, ‘Mind your grammar: a new approach to modelling text, *Proc. VLDB 87P, Brighton (September 1987) 339-346.*

[Jacobs93] P. Jacobs, ‘Integration Frameworks, *SunWorld Magazine* (January 1993) 78-83.

[Julienne93] A. Julienne and L. Russell, ‘Why You Need ToolTalk, *SunExpert Magazine* (March 1993) 50-59.

[Halasz91] F. Halasz, ‘Reflections on NoteCards revisited, presented at *Hypertext 91*, San Antonio (December 1991).

[OpenText93a] Open Text Corporation, *PAT Reference Manual and Tutorial*, 1993.

[OpenText93b] Open Text Corporation, *PATMOTIF User’s Tutorial*, 1993.

[Pearl89] A. Pearl, ‘Sun’s Link Service: A Protocol for Open Linking, *Hypertext ’89*, Pittsburgh (November 1991) 137-146.

[Raymond88] D.R. Raymond and F.W. Tompa, ‘Hypertext and the *Oxford English Dictionary*, *Comm. ACM 31*,7 (July 1988) 871-879.

[Raymond90] D.R. Raymond, ‘Lector — An interactive formatter for tagged text, Tech. Rept. CS-90-34, University of Waterloo (Sept. 1990) 26 pp.

[Raymond91] D.R. Raymond, ‘Reading source code, Tech. Rept. TR 74.070, IBM Canada Laboratory, Toronto, Ontario, (October 1991).

[Raymond92] D.R. Raymond, Flexible text display with ‘Lector, *Computer 25* (August 1992) 49-60.

[Salminen92] A. Salminen and F.W. Tompa, ‘PAT Expressions: an algebra for text search, *Papers in Computational Lexicography: COMPLEX ’92*, Proc. 2nd Int. Conf. on Computational Lexicography (F. Kiefer, G. Kiss, J. Pajzs, ed.), Linguistics Inst., Hungarian Academy of Science, Budapest (October 1992), 309-332.

[Townsend89] G.V.J. Townsend, ‘Citation matching in the *Oxford English Dictionary*, Tech. Rept. OED-89-06, UW Centre for the New OED and Text Research, University of Waterloo (October 1989) 125 pp.