Automated Link Creation in a Hypertext-based Software Reuse Library

Christine L. Tsien
Software and Systems Laboratory
HPL-91-131
September, 1991

software reuse, software reuse library, hypertext, automated linking

The goal of this project is to explore techniques for making software reuse effective. As the development of large software systems continues, the idea of reusing software from previous applications becomes attractive as a means for improving productivity and increasing software quality. Important to such goals is the availability of software components which are well-documented, readily accessible, understandable, and reliable. Software reuse libraries are designed to maintain such components. Because of the large amount of information that may potentially be in a software reuse library, however, a user needs to be able to quickly find, retrieve, and understand a particular software component if reuse is to be fully taken advantage of.

In this project, I aim to make information access, in the context of an existing software reuse library, more efficient. I begin by defining a set of catalogue conventions for the software components of the library. I then develop a system which incorporates these conventions into the library search process. Relevant pieces of information within the library are connected together, or linked, to allow easy reference amongst similar subjects. This is achieved by providing a means for automatically generating hypertext links at library creation time.

Contents

1 Introduction

As technology continues to advance the power of computers, large application software systems are being created. With the development of these software systems comes the notion of reusing those pieces of the software development process which may be similar or unchanged across applications. These software "components" [1] are potentially effective means for increasing programmer productivity and software reliability.[16, 10, 12] This potential benefit is derived from the availability to application developers of well-documented software components which are readily accessible, understandable, and reliable. Maintaining such software components is the goal of a software reuse library.

A software reuse library generally consists of a number of software components which are accessible by users of the library. These components may be documentation, test cases, known bugs, user attempts, etc., as well as source code files. A large set of reusable components needs to be available for reuse to be effective.[3] However, a large amount of information stored in a software reuse library makes the ability to quickly find, retrieve, and understand a particular software component an important factor in reuse effectiveness.[7, 17] When a reuse library is based upon a hypertext [2] system of nodes and links, software reuse can be facilitated by creating links among related pieces of information[16] within that library.

1.1 Motivation for Automatic Linking

In a system containing many items which will be browsed through, connecting related items may be a method for facilitating the retrieval of desired information. In the case of a software reuse library which is based upon a hypertext system, hypertext links are the natural way for implementing such a facility.

By linking related pieces of information in a library, a user can potentially find a desired software component more easily, as well as better understand a component which has already been found. These two advantages are achieved through the existence and the presentation of external links[3] and internal links.[4] During the search for a particular library component, external links can help the user to reach that component from a related component, which may or may not be of the same software module. After a desired software component has been found, the user can follow internal links to better understand and become familiar with the details of this module.

Because reuse libraries can be potentially huge collections of information, the idea of creating external and internal links by hand within such a library quickly becomes infeasible. Instead, a method for *automatically* creating such links becomes attractive.

The existing software reuse library framework which will be the focus of my efforts is

[1] Throughout this document, software "module" will refer to an abstract piece of software, such as a procedure or a particular functionality. Software "components" are tangible parts of a software module, such as source code, documentation, test cases, user attempt reports, and makefiles. A software reuse library is likely to contain several components for any given module.

[2] Hypertext is a technology which allows text files to be interconnected in such a way that one can traverse these connections (referred to as links) from a given text file (referred to as a node) to reach other text files (connected nodes).

[3] External links are those which connect one component to another component within a reuse library.

[4] Internal links are those which connect related pieces of information, residing in possibly different component files, about any one particular software module in a reuse library.

the Kiosk system.[6] This system supports the creation, maintenance, and presentation of reuse libraries whose structures are determined by the specifications outlined in data files.[5] Prior to my work with the system, Kiosk had a method for generating the external, structural links with which to create library networks. My work focuses on developing the automatic generation of both external, cataloguing links, and internal links for Kiosk.

1.2 Previous Related Work

The idea of software reuse has been around since 1969.[3, 6] Of the various efforts which have been made and are being made to increase the cost effectiveness of the software development process, three areas are of particular interest to this project. These areas include hypertext systems for reuse, catalogue and classification schemes for library components, and information retrieval techniques for a reuse library.

Hypertext is one area in which efforts have been made. Frakes and Gandel [8] compare different methods, including hypertext, for representing software components for reuse. Carando [4] suggests that hypertext systems provide certain advantages over other systems for software engineering databases. She claims the advantages to be "easy access to the information; visual representation of project elements and relations; and explicit, nonlinear recording of relations."

In addition to hypertext, efforts in catalogue and classification schemes are of interest. Several different projects have developed their own cataloguing schemes for software components.[9, 16, 2, 14] The purpose of developing a cataloguing scheme for the components of a reuse library is primarily to provide at least a minimal set of information needed by a library user to understand a particular module. The pieces of information which constitute an optimal, minimal set is currently undetermined. A different but related issue involves the classification of the software components in a library.[17, 14] Here, the focus is on how the various components should be related to one another.

Finally, information retrieval techniques for a reuse library are of interest. Monegan [16] discusses component selection and retrieval in general, and describes the method of retrieving information with his ORT tool.[6] Fischer, Henninger, and Redmiles [7] address issues on information access in their system, called CODEFINDER.

1.3 Thesis Organization

The remaining sections present the details of my design and implementation of automatic link generation capabilities for the reuse library system called Kiosk.

Section two presents an overview of the Kiosk system. This section also elaborates upon an existing part of the system, the import tool called Cost++, which becomes fundamental in my link generation process.

Section three describes the development of a set of cataloguing conventions for components of a software reuse library. It first discusses my design of the set of conventions, and then details how I incorporated such a set into the Kiosk system.

[5] Section 2 describes the Kiosk system and the notion of data files in more detail.

[6] Object-oriented Reuse Tool is a tool which provides support for the reuse of object-oriented software.

Section four focuses on my design and implementation of a method by which internal links, to support efficient software reuse, can be automatically generated at library creation time for the Kiosk system.

Section five assesses the effectiveness of the automatic link generation capabilities in promoting efficient search and retrieval of library components in Kiosk. Possible future extensions to the project are also discussed.

2 Background: Kiosk System

This section presents a brief description of the Kiosk reuse library system. Special emphasis is given to describing the existing library import tool, Cost++, which plays a large role in the internal link generation system which I later develop, detailed in Section 4.

2.1 Overview

Kiosk is a system that contains and manages libraries of reusable software components.[6] The Kiosk system uses a tool called Cost++ to import existing software components into a reuse library structure based upon the specifications detailed in input data files.[7] The Kiosk system also provides browsers with which an application developer can interactively search as well as augment a reuse library. Components, in the form of files, are thought of as nodes in a hypertext representation. Hypertext links are used to represent the connections amongst and within files.

2.2 Cost++ Import Tool

The import tool, Cost++, is the means by which a collection of files are linked to one another to form a classification network, or library. Cost++ is also the means by which internal links in the network are to be generated, as discussed in Section 4.

A Kiosk library can be thought of as a tree-like[8] structure with "classification" nodes as the internal nodes and clusters of "hub" and "spoke" nodes as the "leaves". The "leaf clusters" each contain one hub node with possibly multiple spoke nodes attached to the hub. The different types of nodes in a Kiosk library are:

1. **Classification nodes** - These are nodes which make up the internal structure of a library representation. Classification nodes connect to either other classification nodes or to hub nodes. Which nodes are connected to which others depends upon the attributes of each.
2. **Spoke nodes** - These nodes consist of the files corresponding to the components of a software module. Spoke nodes include source code files, header files, makefiles, test cases, documentation files, user attempts, etc.
3. **Hub nodes** - These nodes provide the link between the classification nodes and spoke nodes. Each hub node serves as the catalogue card for a particular software module.[9] When viewed in the library by a user, these nodes contain the name of the software module, a brief description of its function, the computer language in which this module is implemented, and hooks (links) to all other pieces of information which have been established as important in being able to understand and use the module.

Figure 1 shows an abstract representation of how the three node types are related, in the context of a reuse library.

[7] See Appendix A for a sample data file.

[8] The library structure differs from a strict tree structure in that multiple edges (from multiple nodes) can point to a given descendent node. (See Footnote 8 on page 18 for more detail.)

[9] Section 3 describes in more detail the manner in which I modified these hub nodes.

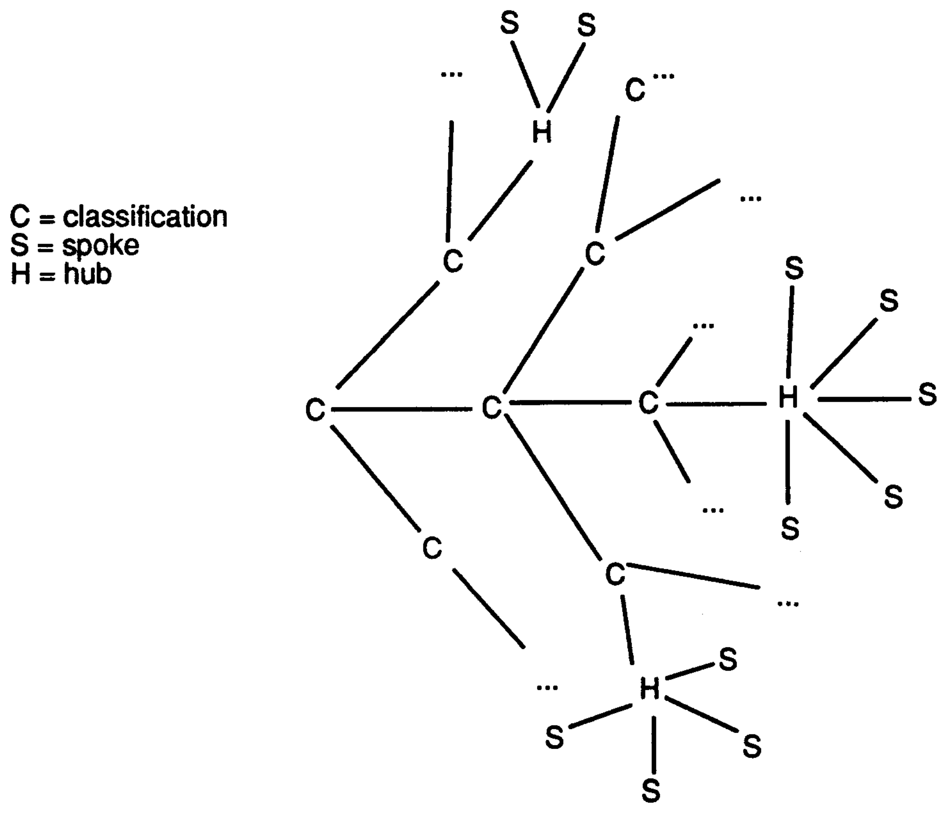

Figure 1: Library File Organization

A data file (given as argument to Cost++) completely specifies the manner in which a corresponding software reuse library is structured.[10] Such a data file is read in during the library-building process (during execution of Cost++). The top of the data file contains a meta-description of the reuse library's structure. A link-description, which fully describes the internal linking scheme of the library[11], has been added to effect the desired linking functionality. The remainder of the data file specifies the classification details of the software components which will make up the actual library network to be created.

2.2.1 Meta Description

The meta-description is located at the very beginning of a data file and is distinguished from the remainder of the file through the use of special delimiters. When Cost++ begins to execute, this meta-description is scanned in, and Kiosk builds what is referred to as an "import model". The meta-description region consists of several description lines. Each description line details an "import model item". These import model items are used for subsequent parsing of the reuse library network description located at the end of the data file. Each description line contains information such as: the type of library component

[10] Section 4 will describe how this data file also completely specifies what types of internal links should be created.

[11] The internal linking scheme is described in Section 4.

for which the current model item is applicable[12], the number of arguments expected for the current import model item, the name given to a link from a hub node to a spoke node, a search string[13], an input directory which specifies where files to be linked can be read, and an output directory which specifies where files which represent hypertext link implementation will be stored.

2.2.2 Link Description

The link-description, which I added to support the link generation scheme, follows the meta-description. It is also delimited in a manner which distinguishes it from the meta and network descriptions. The linking scheme will be described in detail in Section 4.

2.2.3 Network Description

The remainder of the data file specifies the actual structure of a reuse library. Figure 2 shows an example of a small segment of such a data file. Classification nodes, which are "nonleaf" nodes, have the label MAIN and contain names for the following: node, input file, output file, and link. Hub nodes are also considered main nodes of the classification network, and thus, also have the label, MAIN. Hub nodes contain the names of a software module and of its implementation language. Following each hub node are descriptions of the spoke files which connect to the current hub. The spoke files correspond to the software components of the reuse library.[14]

```
                         .
                         .
                         .

[NONLEAF {MAIN "Sets" "" "C++/Sets" "func_view"}
  [HUB {MAIN "Bitset" "C++"}
                         {NROFF_DOC "bitset.3x"}
                         {C++_SRC_CODE "bitset/bitset.c"}
                         {USAGE_ATTEMPT}
                 ]]
[NONLEAF {MAIN "Hashing" "" "C++/Hashing" "func_view"}
  [HUB {MAIN "Hashtable" "C++"}
                         .
                         .
                         .
```

Figure 2: Network Description Format

[12]Example types are: C++ source code, C source code, documentation in nroff form, and documentation in LATEXform.

[13]The specification of a search string provides an anchor for the external, cataloguing links which originate from a hub node.

[14]It is sometimes necessary to refer to a node from multiple locations. This allows for the building of a lattice ("tree-like" structure) instead of only a tree. The import data file can have *references* to objects by using a '#<label>' form which indicates that the object with name <label> is to be referenced at this point, while the actual object is defined elsewhere in the data file. An object referenced by such a label can be defined either before or after the reference.

3 Catalogue Conventions for a Software Reuse Library

This section describes the process by which I developed and implemented a set of cataloguing conventions in the Kiosk system. The motivation for maintaining a cataloguing convention is to have a common basis for any given library module with which to learn about that module and from which to find its related component files. Having such a basis should improve reuse effectiveness by ensuring that at least a minimal amount of useful information is available.

3.1 Defining a Set of Catalogue Conventions

An initial set of catalogue conventions for software reuse libraries has been formulated. Tables 1 and 2 present this cataloguing information. The items in the lists have been preliminarily determined to be useful in assisting application programmers in finding, understanding, and using software components from a reuse library. In establishing such a set, several papers related to software reuse libraries, and cataloging in particular, were referred to for ideas on previous work.[9, 16, 2, 14]

The elements of the lists in Tables 1 and 2 are general enough that they can be applied to other reuse libraries. Some of the terminology with which these elements have been presented, however, is specific to the Kiosk system.[15]

3.2 Incorporating Catalogue Conventions into Kiosk

A few different ideas were considered regarding how to present the catalogue information to Kiosk library users. The most prominent of these ideas were:

1. Convert "hub" nodes into catalogue cards which hold within them all of the relevant information needed for understanding a particular software module. This would involve first finding the desired catalogue information from the various library files (e.g., source code, header, makefile, man page, etc.), and then developing methods of extracting this information in order to insert it in the catalogue card (hub node).
2. Create a separate "catalogue card" file which would hold all of the relevant information as described in the first idea. This catalogue card would then be linked to the hub node.
3. Massage the existing hub nodes into centers, or directories, of information which can then serve as catalogue cards. More specifically, provide hooks (links) to all of the pieces of catalogue information, which reside elsewhere.

After weighing the pros and cons of these alternatives, the third idea was chosen. The deciding factors included ease to the application developer in finding relevant information about reusable software components, and feasibility in relatively rapid prototyping.

[15]For example, the reference to "hub" and "classification" nodes is specific to Kiosk.

Table 1: Information Presented in a Hub Node

```
Hub nodes display:
=================================================
Name of module
Brief overview
Implementation language
-------------------------------------------------

Hub nodes should have links to:
=================================================
Usage Syntax
Synopsis
Portability issues/requirements
File names of relevance
Other commands/options related to module
Author(s)
Performance issues, known bugs, macros used
Version number
Date created
Date last modified
Depends on, inheritance
Size of module, number of source code lines
-------------------------------------------------

Hub nodes have links, if available, to:
=================================================
Man page, documentation
Source code
Makefile
Header files
Examples of use
Test cases
Reviews
Usage attempts
Trouble Shooting; person(s) to contact
User modification/customization provisions
Binaries
Testing documentation  (strategy, etc.)
Detailed documentation
-------------------------------------------------
```

Table 2: Information Presented in a Classification Node

```
 Classification nodes display:
==================================================
 Name of node
 Brief overview/description of the category
       with possible mention of descendant nodes
--------------------------------------------------
```

The first idea listed would potentially result in a cumbersome system due to the volume of information which would be stored in the hub nodes. Such a presentation would make it more difficult to scan a catalogue card quickly, and thus more difficult to readily understand the associated software module. In addition, this approach would require the development of sophisticated extraction tools whose job would be to find *and extract* catalogue card information from within software component files.

The second idea not only has the same disadvantages as the first, but also another: a library user would need to first traverse an extra link simply to get to the catalogue card information, only to then find it poorly presented.

The third option would provide an application programmer with a concise view of all available information on a particular module, as well as with quick access to the file where that information is stored. This approach does not require the development of sophisticated extraction tools to examine catalogue card information within component files because the information need not be *extracted*. The file position where the information begins needs only to be *found*.

Information is searched for in the component files and in the data file.[16] For example, within the "man page"[17] file for a module can usually be found a synopsis, author names, and the names of "related files". Within the hub node, various labels of catalogue information (e.g., SYNOPSIS, AUTHOR, FILES) are placed. When a spoke file is being read in, text-based searches are used to look for the desired pieces of catalogue information. When the places are found at which such information is located in the files, the appropriate links from the hub node file to the spoke files are created.

[16]The implementation language for a particular software module is found from the data file.

[17]On-line manual page for a software module.

4 Internal Linking System

This section presents my design and implementation of the internal linking system for Kiosk. The types of these links can be specified prior to running Cost++, and then instances of each link type are generated during library-creation time. Issues considered during the design phase included expandability and flexibility of the system, as well as feasibility in implementation.

Several possibilities were considered while deciding how to go about creating such a system, particularly the method by which *linking possibilities*[18] would be found. Some of these options included using *etags*[18], *sh*[13], *sed*[13], *awk*[13], *perl*[19], and *egrep*[13]. I decided to hand code the basic linking infrastructure and functionality, and then leave the method for actually finding linking possibilities extremely flexible. This was done by allowing for user-defined *feature extractors*[19] to be used for the text searches which would result in link possibilities within files (internal link possibilities).

4.1 Design Overview

The design for generating internal links at library-creation time consists of three primary parts. First, the desired types of links need to be specified. This includes specifying which of the various library files the searches of each link type should perform upon. Second, the actual routines for performing searches, feature extractors, need to run over the library files to find potential locations for creating internal links. Finally, the resulting linking possibilities need to be analyzed so that actual links can be created where heuristically determined appropriate.

4.1.1 Specifying Link Types

Specification of link types is accomplished during the scanning in of the link-description section found near the top of an import data file. An example of a link-description section is shown in Figure 3.

Each pair of LINK items (one SRC_ITEM description line and one DEST_ITEM description line) is converted into two objects, each holding the search information pertaining to their own particular type of search. All pairs of such LINK items are also assigned a unique ID number (which can simply be the integers in increasing order beginning at zero). These ID numbers will become important later in the process for associating results of SRC_ITEM searches with their matching DEST_ITEM search results. Once these two search objects have been made, they are then put into a list which is sorted according to which model item each searching function is intended for. For example, manner, all searching functions intended for "NROFF_DOC" model items are stored together, while all those intended for "C++_SRC_CODE" model items are stored with one another but separate from the NROFF_DOC functions.

After the entire LINK description section has been scanned, the result produced is a table—each table entry has a string name (corresponding to a model item name) and

[18] Linking possibilities are described in Subsection 4.1.2.

[19] Feature extractors search text files for "interesting" places which may later be potentially linked to related "interesting" places elsewhere in the library. Feature extractors can be built-in functions, programs, or shell scripts.

```
%% Link Description

{LINK

  % Links between a class declaration in a header and that
  % class's member functions in C++ source code.
  {SRC_ITEM      HEADER          "1"       "get_class"}
  {DEST_ITEM     C++_SRC_CODE    "m"       "get_memfuncs"}

  % Links between a function definition in source code and its
  % description in a documentation file.
  {SRC_ITEM      C++_SRC_CODE    "1"       "get_func_src"}
  {DEST_ITEM     NROFF_DOC       "1"       "get_func_docs"}

  % Links between a friend to a class and the class declaration
  % of that friend.
  {SRC_ITEM      HEADER          "m"       "get_friend"}
  {DEST_ITEM     HEADER          "1"       "get_class"}

}
```

Figure 3: Sample LINK Description Within A Data File

each of these table entries is associated with a group of searches. A particular group of searches, therefore, is associated with a model item (which corresponds to a type of software component). Each of these groups of searches will be performed on the files having the particular model item type associated.

4.1.2 Executing Feature Extractors

The second part of automatically generating internal links is the actual searching of the library files, which produces possibilities for link anchors, or linking possibilities. Once the LINK description section has been read in and the searching table built, the network description in the remainder of the import data file is parsed. As each description line is read, its model item name is compared against the model item names stored in the searching table. Should there be a match, all of the searching functions stored in the table with the current model item name will be executed on the library files (software component files) specified by the current description line. In the case where the current model item name is not in the searching table, no searches are performed, and hence, the current library file will not have any internal links. For example, consider the fragment of a network description shown in Figure 4.

```
[HUB {MAIN "Ptyopen" "C"}
     {NROFF_DOC "ptyopen.3x"}
     {MAKEFILE "ptyopen/makefile"}
        .
        .
        .
]
```

Figure 4: Network Description Fragment From A Sample Data File

When the first description line after MAIN is read, the model item name, NROFF_DOC, will be checked against the model item names stored in the searching table. NROFF_DOC is found to be present, so the associated searching functions, which in this case is just *get_memfuncs*[20], are executed on the file, `ptyopen.3x`. Any member functions found will produce link possibilities (described below). On the other hand, when the next description line is read and checked against the searching table, the model item name MAKEFILE is not found. Hence, no search functions will be executed on `ptyopen/makefile`, and thus, this latter file will not have any internal links.

For those model items whose names are in the searching table, the linking positions are determined as follows: Each search function that is executed on the current library file scans the contents of the file for some pre-specified pattern. Whenever an instance of this pattern is found, it is converted into a "unique" identifier string, and the current file position is remembered. At the end of the file, the searching function returns a list of possible link positions and their corresponding "identifier" strings, along with the current file's name.

The returned list of link possibilities is then organized into one of two ***link information tables***. One of these tables is for the results of SRC_ITEM searching functions, while the second is for the results of DEST_ITEM searches. Each position in these link information tables is numbered sequentially, beginning at zero, to correspond to the numbers previously assigned to search objects.[21] For example, link possibilities resulting from a SRC_ITEM search with unique ID number 3 will be stored in the element 3 position[22] of the SOURCE link information table. Likewise, link possibilities resulting from a DEST_ITEM search with unique ID number 0 will be stored in the first position of the DESTINATION link information table.

Inside of each link information table element, link possibilities returned from searches are sorted even further: Those with the same "identifier" string are clustered in a group. Thus, each link information table element is associated with a collection of "groups" of link possibilities. For example, consider the function, *get_memfuncs*, which finds locations of a class's member function definitions, using the class name as the "identifier" string. When

[20] See Appendix B for a sample feature extractor.

[21] These numbers correspond to their array indices for efficiency in accessing table elements.

[22] The element 3 position is actually the fourth table entry due to the fact that the table indices begin at zero.

executed on the file, `bitset.c`, "Bitset" will serve as the identifier string for a new link possibility each time a member function, such as *Bitset::bumpsize*, *Bitset::Bitset*, *Bitset::clear*, and *Bitset::hash*, is defined. Within this same source code file, *Bitsetiter::Bitsetiter* is also defined, so "Bitsetiter" will also serve as an identifier string for a link possibility. The four link possibilities associated with "Bitset" will be grouped together, while the one link possibility associated with "Bitsetiter" will be "grouped" by itself. Both of these groups will be stored with the link information table element which corresponds to *get_memfuncs*.

Each link information table element also stores "one_or_many" data ('1' or 'm') to be used when deciding the manner in which multiple linking possibilities should be interconnected.[23] This information is read from the LINK description data and then stored in the corresponding search information object.

4.1.3 Creating Actual Links

The third and final part of generating internal links is determining which link possibilities should be linked to which other link possibilities. After the entire network description has been read in and the specified library files have been parsed, two link information tables, as described above, have been created. Only comparisons between link possibilities of a given element of the SOURCE link information table and the correspondingly-positioned DESTINATION table element's link possibilities need be performed. The significance of link possibilities having the same table position is that they also have the same unique search ID, which means that they constituted a single LINK pair. For each set of pairs of link possibilities, the SOURCE's identifier strings are checked against the DESTINATION's identifier strings. Anytime there is a match, links from the file positions associated with the SOURCE's current "identifier" should be created to connect with those of the DESTINATION's current "identifier".

At this point, which list of file positions are matched with which other list of file positions has been determined. The next step is to use the "one_or_many" information to determine whether: (1) a particular file position should be linked to one other file position ("1-1"), (2) a particular position should be linked to every position in the other list ("1-m"), (3) every position of the current list should be linked to the one of the other list ("m-1"), or (4) every position of the current list should be linked to every position of the other list ("m-m"). For example, a SRC_ITEM search may be to find class declarations inside of header files, in which case the identifier string is the class name itself, and the "one_or_many" field is '1'. The corresponding DEST_ITEM search may be to find definitions of a class's member functions inside of source code. If the identifier string is to be the *class* name, then this search's "one_or_many" field is 'm' since once class may have many member functions.

Once the internal links have all been created, the library files are then saved out (e.g., to disk), and a new Kiosk reuse library with internal links has been created.

4.2 Implementation Detail

This subsection describes in greater detail the actual implementation of the process explained in the previous subsection.[24]

[23] The use of the "one_or_many" data is described in further detail in Subsection 4.1.3.

[24] This subsection can be safely skipped by the reader without sacrificing loss in overall understanding of the linking system.

4.2.1 Implementation Overview

When a new *Import_model* is created, a new *SearchInfoTable* object is also created as a private member of the former. This *SearchInfoTable* is used to organize all searching functions specified in the LINK description of the input data file. The initialization of a *SearchInfoTable* involves creating and initializing two *LinkInfoTables*, which are to be used later.

During creation of a new *Import_model* object, *Import_model::_open_classification_ data_file* is invoked. This function builds up the contents of the search table by calling *Import_model::_build_search_table*, which then registers all of the searching functions ("feature extractors") into a database (a *Search_func_database*). *_build_search_table* also parses each LINK description line, using *Import_table::_read_search_model_item*, and then creates a *SearchInfo* object from this parsed information. The created *SearchInfo* object is then added to the *SearchInfoTable*, using *SearchInfoTable::add_ search*.

Import_model::read_in_classification_node is soon called from the top level. This function will read in the network description specified in the import data file. Within this function, various other functions are invoked, eventually leading to the execution of *Import_model::dir_assist_general_node_parse_func* upon each file listed in the network description. This latter function not only parses each file to bring it into the reuse library being created, but also calls *SearchInfoTable::do_searches_ find_lspots* with the current import model item and the current file as arguments. *do_searches_find_lspots* is responsible for determining which search functions, if any, should be executed on the current file. This function is also responsible for invoking the functions to search the files and for organizing the resulting information (in the form of *LinkInfo* objects) into a structure which can later be analyzed to determine where to create links. The *LinkInfo*s are stored in the previously mentioned *LinkInfoTables*—one for storing the *LinkInfo*s resulting from searches on "SRC_ITEM" LINK items, and the other for storing the *LinkInfo*s resulting from searches on "DEST_ITEM" LINK items.

Finally, when the entire network description and corresponding library files have been read in, and thus all of the search functions have executed when appropriate to do so, the two *LinkInfoTables* are left holding all possibilities for internal link positions. At this time, *SearchInfoTable::make_internal- _links* is called from the top level function. *make_internal_links* filters through the information in both *LinkInfoTables*: When a *LinkInfo*'s "identifier" string from an element of the SOURCE *LinkInfoTable* matches the "identifier" string of a *LinkInfo* from the corresponding element of the DESTINATION *LinkInfoTable*, then a link is created by means of invoking *LinkInfo::link_LinkSpots*, which in turn calls *link_nodes* to create the actual links.

Descriptions of the data structures, or abstract types, used in implementing the automated linking scheme for Kiosk are located in Appendix C.

4.2.2 Organization of the SearchInfoTable

The *SearchInfoTable* is the framework for part one of the generation of internal links, as mentioned previously. The structure of a *SearchInfoTable* is depicted in Figure 5.

A *SearchInfoTable* is a table of *SITelt* objects. An *SITelt* hold a string name and a pointer to a *SearchInfoList*. Each *SITelt* is associated with the model item whose name matches this *SITelt*'s string field. All *SearchInfo*'s, grouped together in one *SearchInfoList* to which a particular *SITelt* points, represent the various feature extractors which will be performed on library files of the matching model item type. Each SRC_ITEM *SearchInfo* has a unique identifying number, or search_UID, as does each DEST_ITEM *SearchInfo*. SRC_ITEM and DEST_ITEM feature extractors which were specified together as a pair in the LINK description have the same search_UID's.

Appendix C describes each of the portrayed objects and their functions.

4.2.3 Organization of the LinkInfoTable

A pair of *LinkInfoTables* is the framework for part two of the generation of internal links, as described in Subsection 4.1.2. The structure of a *LinkInfoTable* is depicted in Figure 6.

One of these tables is referred to as the "SOURCE" table, while the other is its corresponding "destination" table. ("Source" and "DESTINATION" are used in the same sense as in the LINK description section: the source of a link and the destination of a link.) A *LinkInfoTable* is a table of *LITelt* objects. Each *LITelt* holds a character corresponding to "one_or_many" information and a pointer to a *LinkInfoList*. Each element of a *LinkInfoTable* corresponds to a particular *SearchInfo*. Their correspondence is: the array element index of a given *LinkInfoTable* element is equal to the search_UID of the *SearchInfo* whose function yielded the objects of linking information held by the current *LITelt*.

Appendix C describes each of the portrayed objects and their functions.

4.2.4 Dependencies Among Abstract Types

The diagram in Figure 7 depicts the dependencies among the modules of the internal link generation system. These abstract types are used to facilitate implementation of the automatic linking scheme.

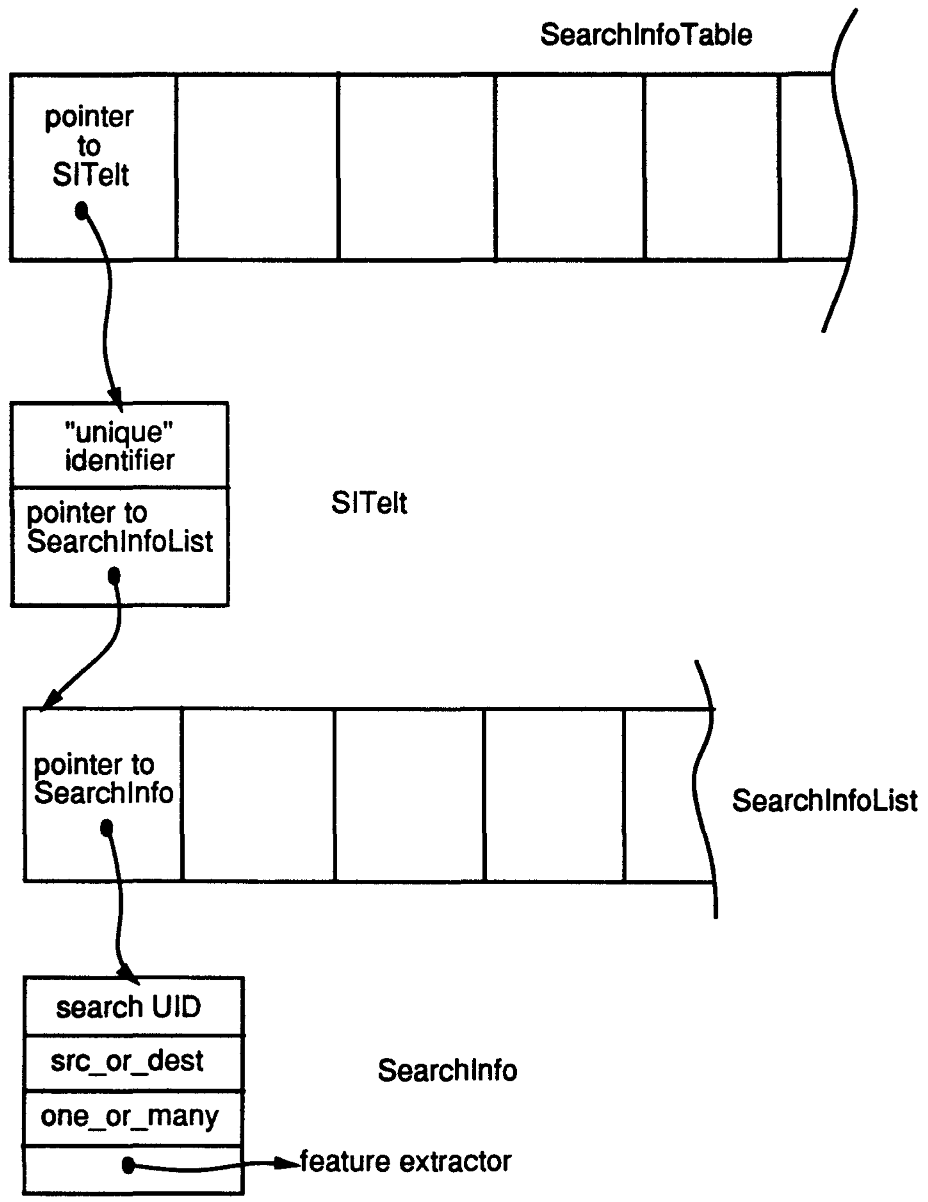

Figure 5: SearchInfoTable Structure

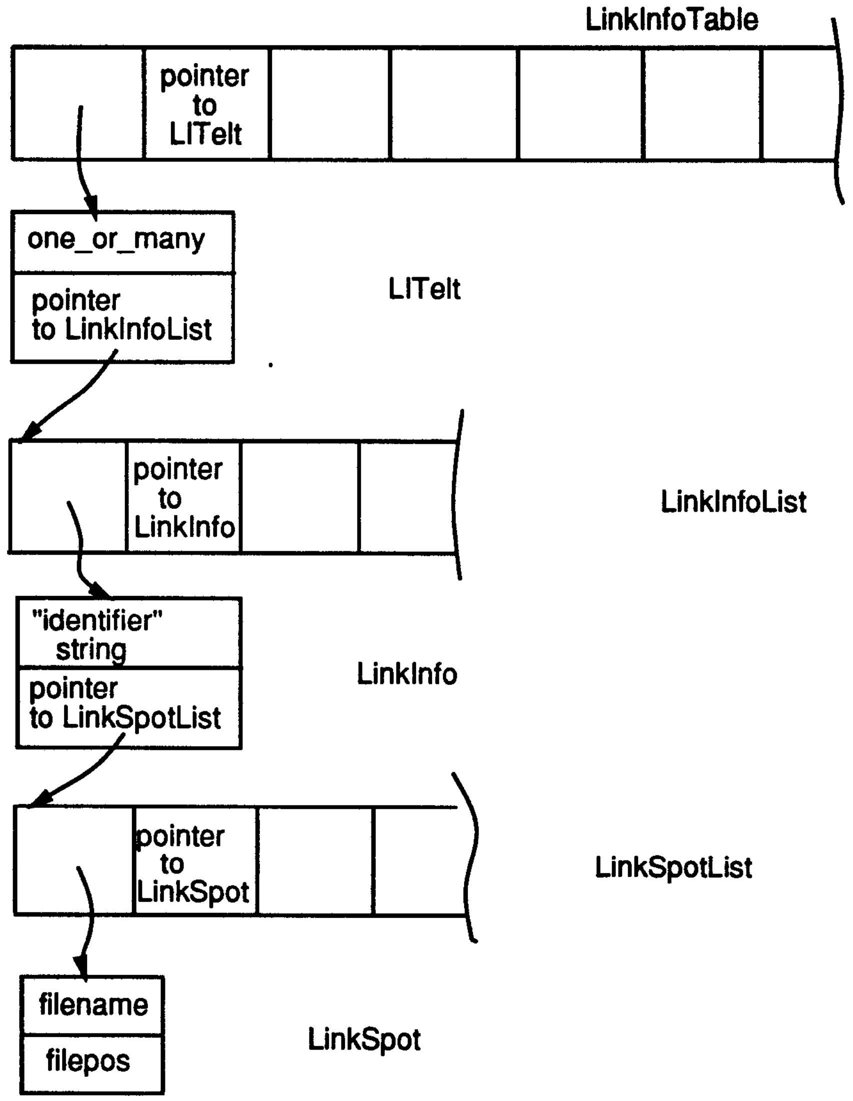

Figure 6: LinkInfoTable Structure

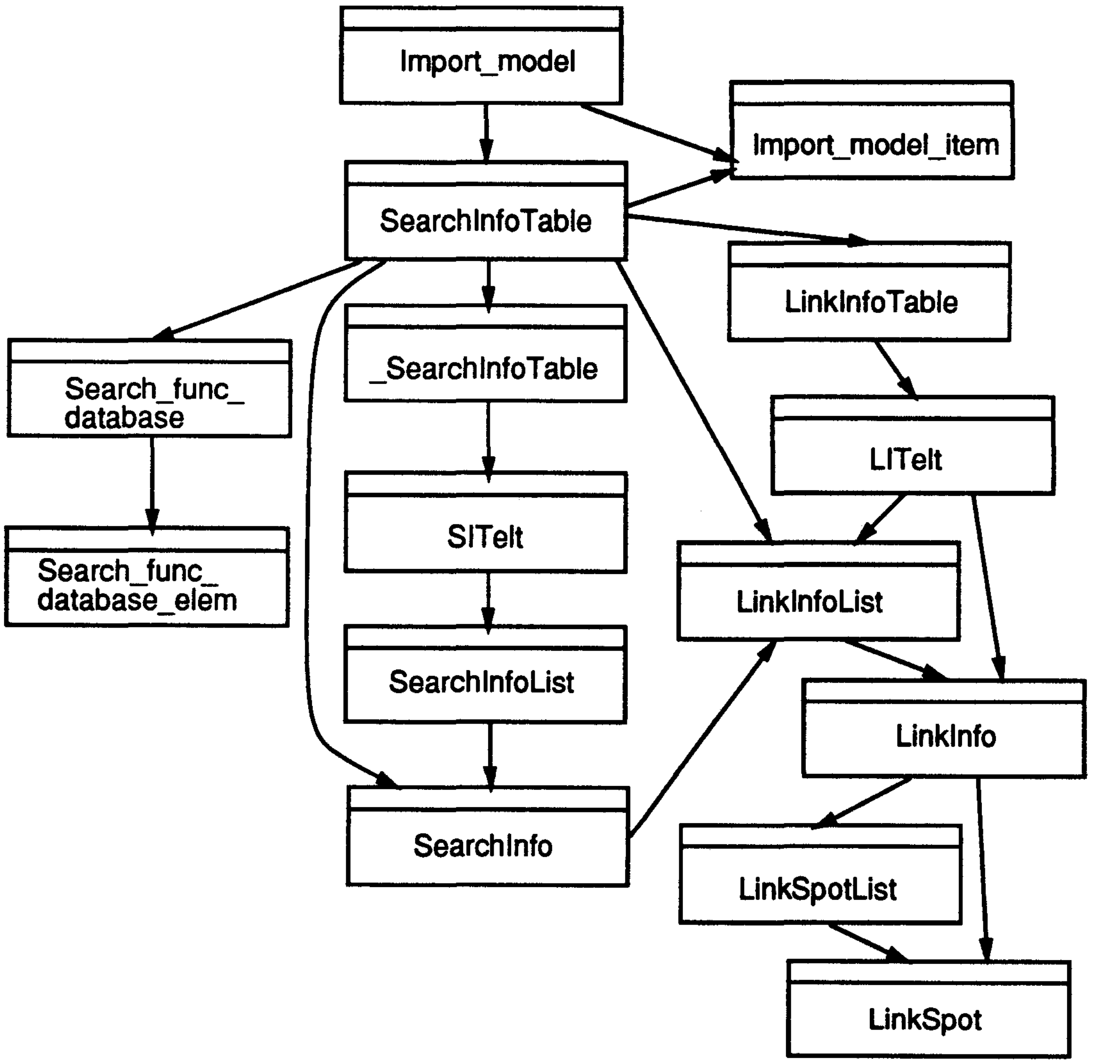

Figure 7: Module Dependency Diagram

5 Summary and Conclusions

In this section, I review the goal and motivation of the project, summarize the project results, and comment on directions for future work.

5.1 Assessment of Link Generation System

The goal of this project has been to improve software reuse effectiveness for the Kiosk library system. Software reuse has been seen as a potential means for improving programmer productivity and increasing software reliability. However, it has also been noted that large collections of software components need to be available if reuse is to be effective. With large component collections, or libraries, though, it becomes more difficult for a user to locate specific information.

In a hypertext-based reuse library system, one attempt at helping a user to locate information involves taking advantage of the nodes-and-links property which is intrinsic to hypertext. The search for and retrieval of information can be facilitated through the establishment of a cataloguing convention scheme and an internal linking scheme. The catalogue conventions have been implemented to ensure that at least a minimal amount of information about each software module in a library is both available and easily found. The internal linking scheme has been implemented to aid in the understanding of how to use a module once it has been found. Because of the potential number of files in a reuse library, the links created for both the cataloguing information and for the internal linking scheme are generated automatically during library creation time.

The effectiveness of incorporating catalogue conventions into a reuse library system will be more readily determined as the Kiosk project progresses and the number of users increases.

The effects of the internal linking system have been monitored during and after the creation of a reuse library. During library creation, the linking functionality has been watched to determine the correctness of the implementation. After the creation process, the library files have been examined to determine whether or not the generated internal links, as specified by the LINK description and the feature extractors, connect useful pieces of information.

The automatic linking functionality works as had been initially conceived. However, not all internal links have been found to correctly connect related pieces of information. This problem may arise due to the heuristic nature of the feature extractors used in determining link possibilities. More cleverly written feature extractors have been observed to produce fewer spurious internal links for the sample reuse library tested. Further use of the Kiosk system will be needed to better determine the advantages and disadvantages of the internal linking scheme.

5.2 Future Work

Without question, there is much additional work to be done to improve a user's ability to find and understand reusable software in a library system. In the sections to follow, I will comment on specific work for the linking scheme of the Kiosk system.

5.2.1 Minor Modifications to Project Work

- Make built-in "feature extractors" (searching methods of class *SearchInfo*) more modular.
- Add the ability to count the number of source code lines when source files are available, and place this information in the HUB nodes. Efficiency may be a concern if *wc -l* [25] is used via system calls. Another concern is in HUB nodes which have more than one source code file: In such cases, the sum of the lines could possibly be reported.
- Allow the META and LINK description sections to be given in any order, with the LINK description's presence entirely optional.
- Move the META and LINK description sections to a separate file so that the same META and LINK specifications can be used for multiple library structures.
- Add an invocation of *SearchInfoTable::do_searches_find_lspots* to *Import_model:: general_node_parse_func* if this is determined as necessary. A hook (commented lines regarding whether or not to add SEARCH functionality) already exists in the code.

5.2.2 Not-so Minor Modifications

- Make the linking system more efficient: If a particular searching function (e.g., *SearchInfo::get_class*) is used by more than one LINK item, the function should only execute once rather than executing the number of times it appears in different LINK items.
- Might want to have the ability to restrict links to files within a HUB rather than to all files in the classification network. A case where this would be useful has already arisen: in linking between class member functions and their definition in documentation, a generic member function name, such as *size*, may be "accidentally" linked to the documentation for that member function name of a *different* class. An idea regarding the implementation for this would be to call the *make_internal_links* procedure after fully reading in each HUB node. This procedure would then be required to clear out the link possibilities listed in the *LinkInfoTable*s upon completion of creating internal links.
- Add an extra parameter in the LINK description for specifying whether a particular internal link type should be performed across the entire library or just within a hub.
- Depending upon presentation of link anchors to a Kiosk user, consider only creating one link to a given file and file position. For example, if there are links from methods of a class to the class definition, there may potentially be numerous methods. If each of the link anchors on the class definition end are presented by their link role name in parentheses, for example, that class definition is likely to look extremely cluttered.
- Consider adding the ability to create links from a position resulting from a search, to a file (e.g. beginning of file), rather than to another searched position.

[25] *wc -l* is a Unix command which gives the "word count" of a file.

- Consider adding the ability to specify "HUB" as a model item to be searched in the LINK description section. This would entail modifying the manner in which the file to be searched is determined. Currently, the file to be searched is the file, named in the network description, associated with the current model item under consideration.

- Develop more ideas for types of internal links to have in a reuse library structure. Add their specifications to the LINK description in a data file.

These ideas are only some of the moderately specific enhancements to Kiosk's linking system. They do not include work on the broader aspects of facilitating information retrieval and understanding in a software reuse library. Clearly, there are more possibilities than have been explored in this project. Nevertheless, the project has examined a number of the issues and ideas related to the use of automatic generation of links to increase reuse software effectiveness.

Acknowledgments

I would like to thank people at HPLabs: Martin Griss, for his support and supervising of my project; Mike Creech, for his technical suggestions, feedback, and project support; Dennis Freeze, for his help with whizzy emacs and X11 commands; Mike Lemon, for his consistent help and advice; and Patricia Collins, for being supportive of my endeavors and introducing me to Kiosk. I would also like to thank various professors at MIT: Peter Elias, for supervising my bachelor's thesis write-up, for being concerned about my wrists, for reading drafts in an excruciatingly precise manner, and for making very good suggestions aimed at making my thesis more readable than this sentence; and Art Smith, for being a very nice person as well as undergraduate advisor, and for supporting me in my many endeavors.

A Sample Data File

The following is a sample data file included to illustrate the layout of the various sections: Meta Description, Link Description, and Network Description. The Network Description is a subset of an actual software reuse library specification.

```
%% Data file for CODELIBS

%% Meta Description

{META
  {MAIN           2 ""               "" "$rlscm"
                    "$krlwc" "no"}
  {VALUE_LINK    -1 ""                 "" "" "" "no"
                    "value_link_parse_func"}
  {DOC           -1 "l-desc"         "Description:<"
                    "$rlscd" "$krlwcd" "no"}
  {LATEX_DOC     -1 "l-desc"         "Description:<"
                    "$rlscd" "$krlwcd" "no"}
  {NROFF_DOC     -1 "l-desc"         "Description:<"
                    "$rlscd" "$krlwcd" "no"}
  {SRC_CODE      -1 "source_code"    "Source Code:<"
                    "$rlscs" "$krlwcs" "no"}
  {C_SRC_CODE    -1 "source_code"    "Source Code:<"
                    "$rlscs" "$krlwcs" "no"}
  {C++_SRC_CODE  -1 "source_code"    "Source Code:<"
                    "$rlscs" "$krlwcs" "no"}
  {HEADER        -1 "header"         "Header File:<"
                    "$rlscs" "$krlwch" "no"}
  {DEPENDS_ON    -1 "depends_on"     "Depends On:<"
                    "" "" "no"}
  {MAKEFILE      -1 "makefile"       "Makefile:<"
                    "$rlscs" "$krlwcs" "no"}
  {CONFIG        -1 "config"         "Building:<"
                    "$rlscs" "$krlwcs" "no"}
  {TEST          -1 "test"           "Tests:<"
                    "$rlscs" "$krlwct" "no"}
  {INHERITS_FROM -1 "inherits_from" "Inherits From:<"
                    "" "" "no"}
  {USAGE_ATTEMPT  0 "usage_attempt" "Usage Attempts:<" "$rlscm"
                    "$krlwcu" "yes" "usage" "generator_parse_func"}
  {REVIEW         0 "review" "Reviews:<" "$rlscm" "$krlwcu" "yes"
                    "review" "generator_parse_func"}
}


%% Link Description
```

```
{LINK
  {SRC_ITEM      HEADER          "1"        "get_class"}
  {DEST_ITEM     C++_SRC_CODE    "m"        "get_memfuncs"}

  {SRC_ITEM      C++_SRC_CODE    "1"        "get_func_src"}
  {DEST_ITEM     NROFF_DOC       "m"        "get_func_docs"}
}


%%  Network Description

[NONLEAF {MAIN "Computer Software" ""
               "$krlwc/Computer_Software" "func_view"}
  [NONLEAF {MAIN "Operating Systems" ""
                 "$krlwc/Operating_Systems" "func_view"}
    [NONLEAF {MAIN "BSD" ""
                   "$krlwc/B.S.D" "func_view"}
      [NONLEAF {MAIN "Compatibility" ""
                     "$krlwc/BSD/Compatibility" "func_view"}

                          .
                          .
                          .
                          .

                       ]]]
  [NONLEAF {MAIN "Computer Languages" ""
                 "$krlwc/Computer_Languages" "func_view"}
      [NONLEAF {MAIN "C++" "" "$krlwc/C++/C++" "func_view"}
        [NONLEAF {MAIN "Data Structures" ""
                       "$krlwc/C++/Data_Structures" "func_view"}
          [NONLEAF {MAIN
             "Tables" "" "$krlwc/C++/Tables" "func_view"}
            [HUB {MAIN "Sorttable" "C++"}
                       {NROFF_DOC "sorttable.3x"}
                       {C++_SRC_CODE "sorttable/sorttable.g"}
                       {VALUE_LINK "OS" "HP-UX/6.5"
                                   "machine" "HP300"}
                       {USAGE_ATTEMPT}
                       {REVIEW}
                       {CONFIG "sorttable/config"}
                       {MAKEFILE "sorttable/makefile"}
                       {TEST "sorttable/QATEST/tst.c"
                             "sorttable/QATEST/tst.std"}
                       {DEPENDS_ON #"$krlwc/Element"}
                     ]
            [HUB {MAIN "Element" "C++"}
                       {NROFF_DOC #"$krlwcd/sorttable.3x"}
                       {USAGE_ATTEMPT}
                       {REVIEW}
                       {CONFIG #"$krlwcs/sorttable/config"}
```

```
              {MAKEFILE #"$krlwcs/sorttable/makefile"}
              {C++_SRC_CODE #"$krlwcs/sorttable/sorttable.g"}
              {TEST #"$krlwct/shellutils/QATEST/tst.c"
                    #"$krlwct/shellutils/QATEST/tst.std"}
            ]]
[NONLEAF {MAIN "Sets" ""
               "$krlwc/C++/Sets" "func_view"}
  [HUB {MAIN "Bitset" "C++"}
              {NROFF_DOC "bitset.3x"}
              {C++_SRC_CODE "bitset/bitset.c"}
              {VALUE_LINK "OS" "HP-UX/6.5"
                          "machine" "HP300"}
              {USAGE_ATTEMPT}
              {REVIEW}
              {CONFIG "bitset/config"}
              {MAKEFILE "bitset/makefile"}
              {TEST "bitset/QATEST/tst.c"
                    "bitset/QATEST/tst.std"}
              {HEADER "bitset/bitset.h"}
            ]]
[NONLEAF {MAIN "Arrays" ""
              "$krlwc/C++/Arrays" "func_view"}
  [HUB {MAIN "Array" "C++"}
              {NROFF_DOC #"$krlwcd/dynarray.3x"}
              {HEADER #"$krlwch/dynarray/dynarray.h"}
              {VALUE_LINK "OS" "HP-UX/6.5"
                          "machine" "HP300"}
              {USAGE_ATTEMPT}
              {REVIEW}
              {CONFIG #"$krlwcs/dynarray/config"}
              {MAKEFILE #"$krlwcs/dynarray/makefile"}
              {TEST #"$krlwct/dynarray/QATEST/tst.c"
                    #"$krlwct/dynarray/QATEST/tst.std"}
              {C++_SRC_CODE #"$krlwcs/dynarray/dynarray.g"
                        #"$krlwcs/dynarray/charbuf.c"}
               ]
  [HUB {MAIN "Dynarray" "C++"}
     {NROFF_DOC "dynarray.3x"}
              {HEADER "dynarray/dynarray.h"}
     {USAGE_ATTEMPT}
              {VALUE_LINK "OS" "HP-UX/7.0"
                          "machine" "HP800"}
              {REVIEW}
              {CONFIG "dynarray/config"}
              {MAKEFILE "dynarray/makefile"}
              {TEST "dynarray/QATEST/tst.c"
                    "dynarray/QATEST/tst.std"}
              {C++_SRC_CODE "dynarray/charbuf.c"
                "dynarray/dynarray.g"}
               ]]
[NONLEAF {MAIN "Strings" ""
```

```
                   "$krlwc/C++/Strings" "func_view"}
      [HUB {MAIN "String++" "C++"}
                 {NROFF_DOC "string++.3x"}
                 {USAGE_ATTEMPT}
                 {REVIEW}
                 {CONFIG "string++/config"}
                 {MAKEFILE "string++/makefile"}
                 {VALUE_LINK "OS" "HP-UX/7.0"
                             "machine" "HP800"}
                 {C++_SRC_CODE "string++/strxx_rep.c"
                               "string++/strxx_str.c"
                               "string++/strxx_sub.c"}
                 {HEADER "string++/string++.h"}
                 {DEPENDS_ON #"$krlwc/Stringx"}
                ]]]]
                         .
                         .
                         .
                         .

[NONLEAF {MAIN "C" "" "$krlwc/C/C" "func_view"}
  [NONLEAF {MAIN "Data Structures" ""
                 "$krlwc/C/Data_Structures" "func_view"}
    [NONLEAF {MAIN "Strings" ""
                   "$krlwc/C/Strings" "func_view"}
      [HUB {MAIN "Mbstring" "C"}
               {NROFF_DOC "mbstring.3x"}
               {USAGE_ATTEMPT}
               {REVIEW}
               {CONFIG "mbstring/config"}
               {MAKEFILE "mbstring/Makefile"}
               {TEST "mbstring/QATEST/tst.c"
                     "mbstring/QATEST/tst.std"}
               {VALUE_LINK "OS" "HP-UX/7.0"
                           "machine" "HP800"}
               {C_SRC_CODE "mbstring/mbschr.c"
                           "mbstring/mbslen.c"
                           "mbstring/mbsrchr.c"}
               {HEADER "mbstring/mbstring.h"}
      ]
      [HUB {MAIN "Stringx" "C"}
               {NROFF_DOC "stringx.3x"}
               {USAGE_ATTEMPT}
               {REVIEW}
               {CONFIG "stringx/config"}
               {MAKEFILE "stringx/Makefile"}
               {TEST "stringx/QATEST/strapp.in"
                     "stringx/QATEST/strapp.std"}
               {VALUE_LINK "OS" "HP-UX/7.0"
                           "machine" "HP800"}
               {C_SRC_CODE "stringx/strapp.C"
```

```
                    "stringx/strbld.C"
           "stringx/strcase.C"
           "stringx/strchg.C"
           "stringx/strcmpi.C"
           "stringx/strdupx.c"
           "stringx/strend.C"
           "stringx/strhash.C"
           "stringx/strnlsx.C"
           "stringx/strpos.C"
           "stringx/strsep.C"
           "stringx/strtest.C"
           "stringx/strtokx.C"
           "stringx/strvec.C"
           "stringx/strwcmp.C"}
          {HEADER "stringx/stringx.h"}
                  ]]
[NONLEAF {MAIN "Tables" ""
              "$krlwc/C/Tables" "func_view"}
  [NONLEAF {MAIN "Symbol Tables" ""
               "$krlwc/C/Symbol_Tables" "func_view"}

                  .
                  .
                  .
                  .

 ]]]]]]
```

B Sample Feature Extractor

This section gives the C++ source code for a sample feature extractor used in the internal linking scheme.

```
LinkInfoList*  SearchInfo::get_func_src (Node *spoke_node)
//
// Requires:  spoke_node represents a C++_SRC_CODE file.
// Modifies:  ---
// Effects:   Performs a search on spoke_node for the names of
//            C++ member functions.  Returns a pointer to a LinkInfoList
//            which holds LinkInfos corresponding to each occurrence
//            of a different member function name which was found.
{
  // Function for model item:  C++_SRC_CODE
  LinkInfoList *return_lil = new LinkInfoList;
  LinkInfo *new_linkinfo;
  LinkSpotList *new_lsl;
  LinkSpot *new_linkspot;
  char *key_string;

  bool ok = spoke_node->setup_search ("::[a-z,A-Z,_]*[ (]");
  if (ok)
    {
     int start_index , end_index;
     do
          {
           start_index = spoke_node->stream_search (end_index);
           if (start_index >= 0)
             {
               char *match = spoke_node->fetch_text (start_index , end_index);

               // The LinkInfoList to be returned can have elements
               // (LinkInfos) that have the same ustring b/c it will
               // be remedied upon return to do_searches_find_lspots

               key_string = strtokx (match, "  :(");
               new_linkspot = new LinkSpot (spoke_node, start_index + 2);
               new_lsl = new LinkSpotList;
               new_lsl->end () = new_linkspot;
               new_linkinfo = new LinkInfo (key_string, new_lsl);
               return_lil->end () = new_linkinfo;
             };
          }
     while (start_index >= 0);
    };
  return (return_lil);
};
```

C Abstract Types

This section details the data structures created to support the linking system described in Chapter 4.

Thirteen abstract data types were created to implement the system of automatically generating links for Kiosk. These data abstractions are: *SearchInfo*, *SearchInfoList*, *SITelt*, *_SearchInfoTable*, *SearchInfoTable*, *Search_func_database_elem*, *Search_func_database*, *LinkSpot*, *LinkSpotList*, *LinkInfo*, *LinkInfoList*, *LITelt*, and *LinkInfoTable*.

The first seven data types were developed in order to keep track of and to use the searching functions that need to be executed on specified model items. The other six data types were developed in order to keep track of information returned from searches in order to facilitate the actual creation of internal links. Several of these data types are either simply arrays of specialized objects, or are those specialized objects themselves.

The following sections give further details about the data abstractions.

C.1 The SearchInfo Data Abstraction

SearchInfo is a class which represents an object that holds information related to performing a search over nodes (files) in order to find positions at which to create internal links. A *SearchInfo* object consists of four fields: (1) an integer corresponding to a unique ID number which is associated with each LINK item, (2) a character (either 's' or 'd' corresponding to whether the particular LINK item was a "SRC_ITEM" or "DEST_ITEM", (3) a character (either '1' or 'm' corresponding to whether the current search is to result in one link position (e.g., a class declaration) or in many link positions (e.g., all methods of a class), and (4) a pointer to the actual function which is to do the searching of a node (file). The class methods provide means to create, destroy, access, and alter a *SearchInfo* object.

The external functions for *SearchInfo* are: *SearchInfo*, *SearchInfo*, *is_src_or_dest*, *is_one_or_many*, *get_function*, *get_search_UID*, *get_class*, *get_memfuncs*, *get_func_src*, *get_func_docs*, *get_friend*, and *merge_LIlists*. As new built-in feature extractors are developed, they will become methods of the *SearchInfo* class.

SearchInfo creates a new one of these objects, while *SearchInfo* destroys (frees) it. *is_src_or_dest* returns the character associated with the current *SearchInfo*'s src_or_dest field. Similarly, *is_one_or_many* returns the character associated with this object's one_or_many field. *get_function* returns a pointer to the actual function which is to do the searching of a node. Examples of such searching functions are *get_class*, *get_memfuncs*, *get_func_src*, *get_func_docs*, and *get_friend*. These functions rely on *Node::setup_search*, *Node::stream_search*, and *Node::fetch_text* in order to find link positions. *get_search_UID* returns the integer corresponding to this object's unique ID number. *merge_LIlists* combines two *LinkInfoLists* into one, returning the merged list.

C.2 The SearchInfoList Data Abstraction

The *SearchInfoList* class is simply an array of pointers to *SearchInfo* objects. The implementation of this array class uses that which is provided by the *codelibs* library.

As used in facilitating link generation, each *SearchInfoList* groups together *SearchInfo*s (searching functions) which are to be performed on a particular import model item (e.g., C++_SRC_CODE, HEADER, NROFF_DOC).

C.3 The SITelt, _SearchInfoTable, and SearchInfoTable Data Abstractions

The *SITelt* class was created to be array element objects for the array class *_SearchInfoTable.* Each *SITelt* object has two fields: (1) a string, called the unique_string, which corresponds to a model item name (e.g., LATEX_DOC, C_SRC_CODE, etc.) and (2) a pointer to a SearchInfoList.

The data abstraction, *_SearchInfoTable*, is simply an array of *SITelt* objects. This array, like the *SearchInfoList*, is implemented using the array class provided in the *codelibs* package.

The *SearchInfoTable* class inherits from *_SearchInfoTable.* The two classes are alike except that the *SearchInfoTable* class also has two *LinkInfoTables*, *source_LIT* and *destination_LIT*, as private members.

C.4 The Search_func_database and Search_func_ database_elem Data Abstractions

The *Search_func_database* class is an array object of pointers to *Search_func_database_ elem* objects. This array is implemented using the array class from *codelibs.* Each object of the *Search_func_database_elem* class has two fields: (1) a string corresponding to the name of a function, and (2) a pointer to the function itself. These functions are the search functions which are specified in LINK items and which will be used to search over files while looking for link positions.

C.5 The LinkSpot and LinkSpotList Data Abstractions

The *LinkSpot* class provides an object used to store the location of a link anchor: (1) a pointer to a node (the file in which the link will be added), and (2) an integer corresponding to the file position where the link will be.

The *LinkSpotList* class is an array of pointers to *LinkSpots*. Again, the implementation of this array class is through the array object provided in the *codelibs* library.

C.6 The LinkInfo and LinkInfoList Data Abstractions

The *LinkInfo* class provides an object used to store information needed when determining which *LinkSpots* should be linked to which other *LinkSpots.* Each *LinkInfo* contains two fields: (1) a string (extracted from a particular searching function), and (2) a pointer to a *LinkSpotList.* The string is used in determining whether or not to create internal links: if the string of a particular *LinkInfo* object matches the string of a corresponding *LinkInfo* object, then the *LinkSpots* contained in the first *LinkSpotList* will be linked to those of the second *LinkSpotList.*

C.7 The LITelt and LinkInfoTable Data Abstractions

An *LITelt* is an object with two fields: (1) a character (either '1' or 'm'), and (2) a pointer to a *LinkInfoList*. The character specifies whether the searching function which corresponds to the current *LITelt* was expected to produce a single *LinkSpot* per uniquely returned string (e.g. the position of a class declaration), or multiple ('m') *LinkSpots* per uniquely returned string (e.g., positions of each member function for a particular class).

The *LinkInfoTable* class is an array of *LITelt* objects. Two of these tables are needed for searching and linking as described. One corresponds to searches and their returned linking information resulting from SRC_ITEM LINK items (as specified in the LINK description), while the other is the analogous table for the information from DEST_ITEM LINK items. The position of an *LITelt* in a *LinkInfoTable* is important—it corresponds to the search_UID originally given to each *SearchInfo* object. Elements of the SOURCE *LinkInfoTable* are paired with the correspondingly positioned elements of the DESTINATION *LinkInfoTable*.

A *LinkInfoList* associated with a given *LITelt* holds all of the *LinkInfos* which have been generated from performing the searching function whose search_UID is the same as the current *LITelt*'s table position.

References

[1] Eric Beser. A Hypertext Reusable Library System White Paper. Technical report, Westinghouse Electronic Systems Group, 1988.

[2] Bruce A. Burton, Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Lauren A. Mayes. The Reusable Software Library. In *IEEE Software*, pages 25–33. Intermetrics, Inc, July 1987.

[3] Gianluigi Caldiera and Victor R. Basili. Identifying and Qualifying Reusable Software Components. In *Computer*, pages 61–70. University of Maryland, February 1991.

[4] Patricia Carando. SHADOW Fusing Hypertext with AI. In *IEEE Expert*, pages 65–78. Schlumberger, Winter 1989.

[5] Brad Cox. Building malleable systems from software 'chips'. *Computerworld*, pages 59–68, March 30 1987.

[6] Michael L. Creech, Dennis F. Freeze, and Martin L. Griss. KIOSK A Hypertext-based Software Reuse Tool. Technical report, Hewlett Packard Laboratories, Palo Alto, CA 94303, March 1991.

[7] Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive Tools for Locating and Comprehending Software Objects for Reuse. Technical report, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Campus Box 430, Bulder, Colorado 80309, 1991.

[8] W. B. Frakes and P. B. Gandel. Representing Reusable Software. Technical report, Software Productivity Consortium and AT&T Bell Laboratories, 1990.

[9] W. B. Frakes and B. A. Nejmeh. An Information System for Software Reuse. In *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, pages 142–151, Holmdel, New Jersey 07733, 1987. AT&T Bell Laboratories.

[10] W. B. Frakes and B. A. Nejmeh. Software Reuse Through Information Retrieval. In *Proceedings of the Twentienth Annual Hawaii International Conference on System Sciences*, pages 530–535, Holmdel, New Jersey 07733, 1987. AT&T Bell Laboratories.

[11] Joseph A. Goguen. Reusing and Interconnecting Software Components. *Computer*, pages 16–28, February 1986.

[12] Martin L. Griss. Software Reuse at Hewlett-Packard. Invited submission to the First International Workshop on Software Reusability, January 7 1991.

[13] Hewlett-Packard Company, Ft. Collins, Colorado. *HP-UX Reference*, September 1989. HP-UX Release 7.0, Volume 1, Section 1. awk: pages 95–97; egrep: pages 355–356; sed: pages 627–629; sh: pages 630–639.

[14] Wayne Lim and Sylvan Rubin. Guidelines for Cataloging Reusable Software. Unpublished; Hewlett Packard and Ford Aerospace.

[15] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. Automatically Generating Software Libraries without Pre-Encoded Knowledge. Technical report, IBM Thomas J. Watson Research Center, 1989.

[16] Michael D. Monegan. An Object-Oriented Software Reuse Tool. A.I. Memo 1118, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, April 1989.

[17] Ruben Prieto-Diaz and Peter Freeman. Classifying Software for Reusability. In *IEEE Software*, pages 6–16. GTE Laboratories and University of California at Irvine, 1987.

[18] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, sixth edition, March 1987. Version 8, Section 21.11, pp. 147-152.

[19] Larry Wall and Randall L. Schwartz. *Programming perl.* O'Reilly and Associates, Inc., Sebastopol, CA, 1991.