An Information Retrieval Approach For Automatically Constructing Software Libraries

Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser

Abstract—**Although software reuse presents clear advantages for programmer productivity and code reliability, it is not practiced enough. One of the reasons for the only moderate success of reuse is the lack of software libraries that facilitate the actual locating and understanding of reusable components. This paper describes a technology for automatically assembling large software libraries which promote software reuse by helping the user locate the components closest to her/his needs. Software libraries are automatically assembled from a set of unorganized components by using information retrieval techniques. The construction of the library is done in two steps. First, attributes are automatically extracted from natural language documentation by using a new indexing scheme based on the notions of lexical affinities and quantity of information. Then a hierarchy for browsing is automatically generated using a clustering technique which draws only on the information provided by the attributes. Thanks to the free-text indexing scheme, tools following this approach can accept free-style natural language queries. This technology has been implemented in the GURU system, which has been applied to construct an organized library of AIX utilities. An experiment was conducted in order to evaluate the retrieval effectiveness of GURU as compared to INFOEXPLORER a hypertext library system for AIX 3 on the IBM RISC System/6000 series. We followed the usual evaluation procedure used in information retrieval, based upon recall and precision measures, and determined that our system performs 15% better on a random test set, while being much less expensive to build than INFOEXPLORER.**

Index Terms—**Automatic indexing, clustering, information retrieval, lexical affinities, software libraries, software reuse.**

I. Introduction

Software reuse is widely believed to be a promising means for improving software productivity and reliability [14], and therefore is an issue of growing interest in software engineering. Unfortunately, not enough *adequate* libraries of reusable software components are available. By adequate, we mean that the library:

- Provides a sufficient number of components, over a spectrum of domains, that can be reused as is (*black-box* reuse) or easily adapted (*white-box* reuse)
- Is organized such that the existing code closest to the users' needs is easy to locate. In particular, the library should provide mechanisms to help the reuser look for "functionally close" components which meet some given requirements.

This paper is concerned with the second adequacy issue, and more generally with library systems which provide means for representing, storing, and retrieving reusable components.

The first stage in building a library consists of *indexing* the objects to be stored in it, that is, producing a set of characterizing attributes, or *profile*, for each of these objects. The profile for each object represents the reusable object. Therefore the quality of indexing is crucial to the quality of the library. Functionality is an important aspect of software components. Thus it is necessary to include conceptual information about functionality in the indices. Unfortunately, conceptual information is difficult to obtain. Few programmers provide conceptual indices for their code. Moreover, even if provided, they can hardly be expressed under a common formalism, since pieces of code typically originate from multiple sources. One solution is to manually index software components *a posteriori* according to a given classifying scheme, but this task is expensive.

As an alternative, we propose to automatically identify indices by analyzing the natural-language documentation, in the form of manual pages or comments, usually associated with the code. Natural-language documentation is clearly a rich source of conceptual information. However, this information is contained only implicitly, in an unstructured way, and is not usable as such. In order to extract usable information from free-style documentation we propose to use information retrieval techniques. Once the indices have been produced, components can be automatically classified, stored, and retrieved according to their profiles.

The classifying stage in the construction of a library consists of gathering objects into classes such that the members of the same class share some set of properties. The basic motivation for classifying is to facilitate browsing among similar components in order to identify the best candidates for reuse, or at least a set of potentially adaptable components that can be easily located. Browsing is more important for software libraries than for other kind of libraries, since there rarely exists a component perfectly matching a user's query. Moreover, local browsing allows the user to discover unanticipated opportunities for reuse.

We have designed and implemented a tool, GURU, that

Manuscript received August 28, 1989; revised March 25, 1991. Recommended by D. Barstow. Y. S. Maarek was partially supported by a Gutwirth Fellowship. G. E. Kaiser was supported by the National Science Foundation through Grants CDA-8920080, CCR-8858029, and CCR-8802741, and through Grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun, and Xerox, and by the Center for Advanced Technology and the Center for Telecommunications Research.

Y. S. Maarek is with the IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

D. M. Berry is with the Department of Computer Science, Technion, Israel Institute of Technology Haifa, 32000, Israel.

G. E. Kaiser is with the Department of Computer Science, Columbia University, New York, NY 10027.

IEEE Log Number 9101141.

embodies the above approach. GURU automatically assembles conceptually structured software libraries from a set of unindexed and unorganized software components. In the first stage, GURU extracts the indices from the natural language documentation associated with the software components to be stored by using a new indexing scheme. This indexing scheme is based on *lexical affinities* and their statistical distribution. It identifies a set of attributes for each document to represent a functional description of the associated software unit. In the second stage, GURU assembles the indexed objects into a browsing hierarchy by using a *hierarchical clustering* technique which draws information exclusively from the indices identified in the previous stage. Thus GURU supports both classical linear retrieval, in which candidates are ranked according to a numerical measure that evaluates how well they answer the query, and cluster-based retrieval in which the browse hierarchy directs the search for the best candidate.

Section II briefly compares the artificial intelligence and information retrieval approaches to construction of software libraries and explains why we follow an IR approach. Section III describes the indexing method. Section IV presents the classification approach and the clustering technique used for assembling the library. Section V deals with the retrieval stage. Section VI gives results using our GURU implementation and a formal evaluation based on usual methodology for evaluating information retrieval systems. Finally, Section VII summarizes the main contributions of this work. Related work is discussed as relevant throughout the paper.

II. AI Versus IR Approach

Previous efforts for building reuse systems can be roughly classified into two groups according to the approach[1] adopted, the free-text indexing approach as defined in information retrieval (IR), and the knowledge-based approach as defined in artifical intelligence (AI).

The IR free-text approach[2] consists of drawing information only from the structure of some documents which provide information about the software components. No semantic knowledge is used and no interpretation of the document is given. The reuse tool attempts to characterize the document rather than understand it. There are currently very few software library systems that follow such an approach or use existing IR techniques. Among them, the RSL [6] system, for instance, automatically scans source code files and extracts comments explicitly labeled for reuse with attributes such as keyword, author, date created, etc. The keyword attribute provides a list of free-text single-term indices very much like those used in IR tools. The REUSE [3] system provides a menu-driven front end to an information-retrieval system. Thus all kinds of software objects, including user menus and system thesauri, are stored as textual documents. These two systems use some kind of IR-related technique. However, the only system, to our knowledge, which applies a pure IR free-text approach is the system proposed by Frakes and Nejmeh [16]. They use the CATALOG information-retrieval system for storing and retrieving C software components. Each component is characterized by a set of single-term indices that are automatically extracted from the natural-language headers of C programs. Therefore the construction of the C components repository is done automatically and does not require any pre-encoded knowledge, as in RSL, for instance.

In contrast, in the knowledge-based approach the reuse tool aims at understanding the queries and functionality of components before providing an answer. Knowledge-based systems are often smarter than IR systems. Some of them are context sensitive and can generate answers adapted to the user's expertise. As a trade-off, they require some domain analysis and a great deal of pre-encoded semantic information, which is usually provided manually. They are based upon a knowledge base which stores semantic information about the domain and about the language itself in case of a natural-language interface. The main problem of applying this approach in the context of software libraries is that many domains cannot be easily circumscribed and the domain analysis is very difficult [10]. This makes the construction of such systems very tedious and expensive. Examples of AI or knowledge-based reuse tools are numerous; e.g., [32], [41], [2], [11], [39].

The AI approach can be useful in some applications. However, we prefer the IR approach for reasons of:

- Cost: the library system is built entirely automatically
- Transportability: the library system can be rebuilt for any domain, since it does require manually provided domain knowledge
- Scalability: the repository can be easily updated when new components are inserted, either by recompiling the indices or by applying incremental techniques; the indexing task is entirely mechanical.

We therefore apply a pure IR approach, in the same direction as that of Frakes and Nejmeh, by automatically building free-text indices that characterize software components. For more effective retrieval we also use a free-text method which is richer than the single-term indexing used in the IR-based tools described above. The following section explains our source of information and how the indexing is performed.

III. The Indexing Stage

The major advantage of automatic indexing over manual indexing, besides the obvious cost considerations, is that it allows a unified scheme which ensures that indices will be compatible with each other. The idea is to extract attributes from an existing source of information; i.e., the code and natural-language documentation. Some work has been done toward extraction of primitive functional information from the code [28], [36]; however, the richer source of functional information is the natural-language documentation, assuming that any is available.

An examination of numerous samples of code allowed us to reach the conclusion that some useful information can be

[1]Another approach is the hypertext approach (see [15] for a survey). We do not address this approach here because we are concerned with the type of information used to *build* a library rather than with searching. The hypertext approach is orthogonal to the approaches described here; and hypertext tools can easily be integrated with most IR- or AI-based reuse tools.

[2]For brevity we will refer to this approach as the IR approach even though some IR techniques do not use free-text indexing.

extracted from programs written in a high-level language using good programming style, whereas little conceptual information can be found in typical real-world code chosen at random [26]. Unfortunately, even when dealing with well-written code, there is a very low probability that the programming styles of the various pieces of code will be consistent. Even a single programmer may use totally different identifiers for expressing the same concept from one day to another. Since software components come from multiple sources in the context of large software libraries, extracting attributes from code would necessitate as many indexing schemes as there are code sources. Another limitation comes from the fact that there are many more possibilities for identifiers than for natural-language words, since they do not follow any morphological or syntactic rules.

In other words, when there is no way to guarantee good, let alone consistent and compatible, programming styles, extracting attributes from raw code does not give significant results. Therefore we prefer concentrating on the other possible source of information; i.e., the natural-language documentation either inserted into the code— the comments—or associated with the code, e.g., manual pages.

Comments are intended to help programmers understand the code and thus may provide functional information. They deal with specific parts of the code into which they are inserted and they may give information on various parts at various levels of abstraction. Extracting functional information from comments entails two activities:

- Defining an indexing scheme which allows extracting attributes from natural language phrases or sentences
- Relating comments to the portion of code they concern.

The second activity is very complex in free-style code. Indeed, in free-style programming, programmers can insert comments wherever and in any format and any length they wish. Although comments usually describe the containing routine or the one just below, in general it is impossible to automatically determine what part of the code is covered. A solution would be to consider that all the comments inserted in a specific piece of code constitute a global natural-language description of the considered code. Unfortunately, this is not the case. Comments rank from low-level implementation details to high-level description. For instance, in the `rm.c` source file in Berkeley UNIX, one can find comments as various as:

```
/* current pointer to end of path */, or
/* rm - for ReMoving files, directories & trees.
*/
```

The first conveys no useful functional information, while the second hits the mark exactly. In general, there are many more low-level—and useless for our purpose—comments than high-level ones, and there is no way to automatically distinguish between them. Therefore, so long as no style is enforced, it is very difficult to extract useful information from comments.

Let us note, however, that any piece of natural language, from comments inserted in the code to design specifications, that is specifically related to software code and whose level of abstraction is known can bring useful information. Thus we are currently working on extracting functional information from comments in the framework of RPDE [18], a structured software development environment, in which comments are linked to the portion of code they describe. In the following, though, we try to remain as general as possible and we do not assume that any commenting style is enforced. Therefore, although our indexing scheme is applicable to any piece of natural-language that brings some functional information, we will exemplify it through the analysis of manual pages clearly related to reusable components, such as UNIX-like manual pages.

In the rest of this paper the AIX documentation is taken as our corpus, since it fulfills the requirement of being structured into manual pages. Moreover, the AIX documentation can be seen as a regular real-world documentation database, since it is of average quality as far as commenting style is concerned. Many even consider the AIX documentation of poor quality when compared to Berkeley UNIX documentation due to typos, inconsistent style, poor vocabulary, etc.

A. A Richer Indexing Unit: The Lexical Affinity

There has been much work in IR dealing with natural-language text: a large variety of techniques have been devised for indexing, classifying, and retrieving documents [33], [34]. One of the main concerns in IR is the automatic indexing of documents, which consists of producing for each document a set of indices that form a *profile* of the document. A profile is a short-form description of a document, easier to manipulate than the entire document, that plays the role of a surrogate at the retrieval stage.

Several issues need to be addressed when indexing a document with respect to the nature and form of the produced indices. More precisely, the indexing vocabulary can be either controlled or uncontrolled. In the controlled vocabulary approach only a restricted set of indices are authorized (for example, in MEDLARS [34]), whereas in the uncontrolled vocabulary or free text approach, there is no constraint on the nature of the indices. It has been shown that both approaches are comparable in terms of performance [14], [34]; however, we prefer the uncontrolled vocabulary approach in the context of software reuse for the same reasons of cost, portability, and scalability. Indeed, defining an adequate controlled vocabulary is a manual domain-dependent task and therefore suffers from the same drawbacks as the encoding of a knowledge-base.

Another important issue in automatic indexing is the nature of the indices. The most usual form is a single-term index, each of which is a single word without contextual information. It has also been proposed to use term phrases as indexing units rather than single terms so as to refine the meaning of constituent words. However, the use of word co-occurrences has not brought good results. As expressed by Salton [33, p. 296]:

> " . . . a phrase-formation process controlled only by word co-occurrences and the document frequencies of certain words is not likely to generate a large number of high-quality phrases."

A possible solution to this problem is to use syntactic information such as part-of-speech derived from specially formatted dictionaries [23] in order to provide further control over phrase formation or more refined analysis including semantics [38]. But [33, p. 298]:

> "The available options in phrase generation appear limited, and the introduction of costly and refined methodologies may bring only marginal improvements."

We are more optimistic and believe that indexing units richer than single terms can be used, and that they can bring significant improvement at low cost. The atomic unit we propose to use in order to demonstrate this is derived from the notion of *lexical affinity*. In linguistics, a syntagmatic lexical affinity (LA), also termed a *lexical relation*, between two units of language stands for a correlation of their common appearance in the utterances of the language [8]. The observation of LA's in large textual corpora has been shown to convey information on both syntactic and semantic levels and provides us with a powerful way of taking context into account [37].

We propose to use the notion of LA for indexing purposes and restrict the above definition by observing LA's within a finite document rather than within the whole language so as to retrieve *conceptual* affinities that characterize the document rather than purely *lexical* ones. Moreover, we only consider LA's involving *open-class words* as meaning-bearing, whereas LA's involving *closed-class* words[3] are not.

Ideally, LA's are extracted from a text by parsing it, since two words share a lexical affinity if they are involved in a modifier-modified relation. Unfortunately, automatic syntactic parsing of free-style text is still not very efficient [35]. Instead, we make use of simple co-occurrence. It has been shown by Martin *et al.* that 98% of lexical relations relate words which are separated by at most five words within a single sentence [30]. Therefore most of the LA's involving a word w can be extracted by examining the neighborhood of each occurrence of w within a span of five words (-5 words and $+5$ words around w).

The extraction technique consists of sliding a window over the text and storing pairs of words involving the head of the window (if it is an open-class word) and any of the other open-class elements of the window. The window is slid word by word from the first word of the sentence to the last, the size of the window decreasing at the end of the sentence so as not to cross sentence boundaries[4], since lexical affinities cannot relate words belonging to different sentences. The window size being smaller than a constant, the extraction of LA's is linear in the number of words in the document. An algorithm for the sliding window technique is presented in Fig. 1. Maarek and Smadja have used a similar technique in [29], which was also based on Martin's results [30], but more adapted to the analysis of large corpora.

```
For each sentence $S$ in the document $d$
    For each word $w$ in $S$ from the beginning to the end of $S$
        $w \leftarrow \mathrm{lemma}(w)$
        (where lemma($w$) represents the inflectional root of $w$)
    EndFor
    For each lemma $w$ in $S$ from the beginning to the end of $S$
        If $w$ is an open-class word then
          Let $w_1, \ldots, w_n$ be the $n$ words immediately following $w$ in $S$
          (where $n = 5$ except when the end of the sentence is reached earlier)
          For $i = 1$ to $n$
              If $w_i$ is an open-class word then
                Get $f$. frequency count of $\{w, w_i\}$
                ($f = 0$ when the LA has not been encountered before)
                Store $\{w, w_i\}$ with a frequency count of $f + 1$
              EndIf
          EndFor
        EndIf
    EndFor
EndFor
```

Fig. 1. Sliding window technique.

In summary, the first stage in indexing a manual page consists of extracting all the potential LA's by using the sliding window technique. Once extracted, the potential LA's are stored under their canonical form, in which each word is represented by its inflectional root, or lemma, i.e., the singular form for nouns and the infinitive form for verbs. The potential LA's extracted from the manual page of mv in AIX and ranked by frequency of occurrence are presented in Table I. For the sake of the comparison, a list of the single words extracted from the same manual page is shown in the first column, also ranked by frequency of appearance.

Among the extracted lexical relations, some correspond to abstractions of the considered document and some do not. In a first stage, we isolate actual affinities by using frequency criteria. It has been demonstrated that the frequency of occurrence of a term within a document is related to its importance in the text [25]. This is also true for the common appearance of pairs of words and even more for lexical affinities.

B. From LA's to Indices

When analyzing a document, many potential lexical affinities are thus identified. Some of these lexical affinities are conceptually important and some are not. As seen in Table I, frequency of appearance is a good indicator of relevance. However, some noise exists, mainly due to words appearing too often in a given context. In order to reduce the influence of such words it is necessary in the second stage to select from among the lexical affinities identified only the most representative ones; i.e., those containing the most information.

We have defined a measure evaluating the *resolving power* of an LA. It is based upon the quantity of information of each of the words involved in the LA as well as upon the frequency of appearance of this LA within the considered document. The *quantity of information* of a word within a corpus is defined as:

$$\mathrm{INFO}(w) = -\log_2(P\{w\}) \qquad (1)$$

where $P\{w\}$ is the observed probability of occurrence w in the corpus [4], [34]. Therefore the more frequent a word is in a domain, the less information it carries. From this definition we infer the definition of the quantity of information of an

[3] In general, open-class words include nouns, verbs, adjectives, and adverbs, while closed-class words are pronouns, prepositions, conjunctions, and interjections.

[4] The isolation of sentences is the only parsing performed.

TABLE I
KEYWORDS AND LEXICAL AFFINITIES CLASSIFIED BY FREQUENCY IN THE mv MANUAL PAGE

Open-class Words	Frequency	LA's	Frequency
file	30	file move	9
directory	14	be file	8
mv	11	directory file	7
files	8	file system	5
new	7	file overwrite	5
name	7	file mv	5
move	7	file name	4
newname	6	name path	3
is	6	do file	3
system	5	directory move	3
one	5	different file	3
...	...	...	...

TABLE II
COMPARISON OF FREQUENCY AND ρ-VALUE FOR THE LA's IN mv

LA's	Frequency	LA's	ρ
file move	9	file move	8.38
be file	8	file mv	4.36
directory file	7	directory file	4.03
file system	5	file overwrite	3.87
file overwrite	5	directory move	1.98
file mv	5	file system	1.95
file name	4	mv rename	1.71
name path	3	move mv	1.58
do file	3	different file	1.40
directory move	3	name path	1.33

LA (w_1, w_2) as:

$$\text{INFO}((w_1, w_2)) = -\log_2(P\{w_1, w_2\}). \quad (2)$$

To simplify the computation of this factor in the rest of this work, we consider words within the textual universe as independent variables.[5] Thus we use the following formula for computing the quantity of information of an LA:

$$\text{INFO}((w_1, w_2)) = -\log_2(P\{w_1\} \times P\{w_2\}). \quad (3)$$

Then we define the resolving power of an LA in a given document as follows: Let (w_1, w_2, f) be a tuple retrieved while analyzing a document *d*, where (w_1, w_2) is an LA appearing f times in *d*. The *resolving power*[6] of this LA in *d* is defined as:

$$\rho((w_1, w_2, f)) = f \times \text{INFO}((w_1, w_2)). \quad (4)$$

The higher the resolving power of a lexical affinity is, the more characteristic of the document it is. The resolving power allows us to evaluate the importance of a lexical affinity within a text by taking into account both its frequency of appearance in the text and the quantity of information of the words involved. Thus, even though the lexical affinity `(be file)` appears very often in an AIX manual page, it has only a small resolving power, simply because the quantity of information of both the words "file" and "be" in the AIX documentation is low.

In order to be able to compare the relative performances in terms of resolving power of different documents, we transform the raw ρ score into a standardized score. The standardized score, or z-score, is defined as $\rho_z = (\rho - \overline{\rho})/\sigma$, where $\overline{\rho}$ and σ are the average and standard deviation of the ρ-values. This transformation does not alter the distribution and allows us to evaluate the relative status of the score in the ρ distribution. In the rest of this paper, the ρ-values we give as examples will therefore represent the z-score rather than the raw score.

Table II compares the list of LA's for the mv manual page ranked by frequency and resolving power. In it, the LA *(file move)* has a greater resolving power than any of the following LA's. Moreover, some noisy LA's such as *(do file)* or *(be file)* (in italic fonts in the table) have disappeared because *both* words involved in the LA's are highly frequent in the corpus and thus have a low quantity of information.

For each document, we select as indices those LA's with the highest resolving power. More precisely, we are interested in the LA's which represent peaks in the distribution of ρ-values. Therefore we keep as indices only the LA's whose ρ value is one standard deviation above the mean; i.e., such that $\rho \geq \overline{\rho} + \sigma$, where $\overline{\rho}$ represents the mean and σ the standard deviation of the distribution of ρ values within one document.

[5]This assumption represents only an approximation, since words in English are definitely not independent but are distributed according to the rules of the language.

[6]This notion is related to that of mutual information [4].

TABLE III
LA's RANKED BY ρ-VALUES FOR cp

LA's	ρ_z
copy file	6.49
directory file	2.47
file source	2.15
infile subdirectory	1.98
contain subdirectory	1.30
copy cp	1.30
copy regular	1.02

The choice of such a threshold[7] is reflected in Tables II–IV, where only LA's with a z-score greater than 1 are presented.

The set of LA's of a document, selected by ranking ρ-values and taking those one standard deviation above the mean, forms the profile of the document. The major contribution of this technique consisted in adapting the notion of lexical affinity for indexing purposes. We gave some intuitive indications on how an LA-based indexing scheme is richer than a single-word scheme. We will demonstrate later that it ensures a better retrieval effectiveness.

The next section explains how software components can be stored and classified using the profiles produced at the indexing stage.

IV. The Classifying Stage

Normally, when a user wants to use a software library, he/she first has to access a library which might contain the desired component, then has to provide a formal description of the researched component according to the vocabulary understood by the library system. Unfortunately, in most cases this ideal scenario does not work out. The main reason is that in real life applications the component perfectly matching the user's requirements does not exist in the library, or that it is not indexed as the user had guessed it would be.

In such cases, a traditional database management system fails to help the user. Indeed, to be retrieved from the database, a component must exactly match the query.[8] Such strict matching is inappropriate in a software library system, since the user often cannot know the exact characteristics of the desirable component and, even when he/she does, there is rarely a perfect match.

[7]This classical threshold guarantees to keep only a small percentage of the sample elements in most distributions.

[8]A notable exception is ARES [20], a relational database which allows flexible interpretation of queries. In ARES, the similarity between elements can be evaluated via a lookup in a table that has to be provided beforehand. ARES is not discussed here, since its purpose is not to classify software. Further, it has the drawback of requiring a great deal of pre-encoded knowledge.

TABLE IV
LA's RANKED BY ρ-VALUES FOR mkdir

LA's	ρ
directory make	5.08
create mkdir	2.74
directory mkdir	2.55
directory permission	1.48
directory write	1.03

Software libraries should not only permit retrieving candidate components which perfectly or partially match the query, but also permit browsing among components that share some functionality. It is therefore desirable to structure the library for making the search, retrieval, and browsing mechanisms as fast and convenient as possible in order to make the access to the library attractive.

We propose here to perform the search and retrieval operations using a conventional inverted index file structure, and to cluster the library in order to facilitate the browsing operation. Section IV-A explains how the index repository is built using an inverted file structure, and Section IV-B presents the clustering technique used to build the browse hierarchy. Section V explains how they are used to perform the search and browsing operations.

A. Building the Index Repository

The goal is to allow fast and easy identification of candidate components during retrieval. Thus an inverted file index is derived from the profile repository built during indexing. Index LA's are defined as tuples (w, w') in which w precedes w' in the lexicographic order. The reason for ordering w and w' is to avoid duplicate LA's by forcing every LA into a canonical form. Moreover, we also store w and w' as individual indices in order to detect partial matching, only one word in common, between query LA's and document LA's.

Every index points toward a list of pairs (d, ρ) in which d is the document whose profile contains the index and ρ is its corresponding normalized resolving power. The information associated with each index is accessed through a trie data structure. Using a trie data structure is advantageous in our case because of the numerous repeated prefixes.

The stored information is used to retrieve and rank candidates as explained in Section V.

B. Building the Browse Hierarchy

As explained previously, browsing is crucial in software library systems. The most common way to make browsing operations possible is to group items judged to be similar by using clustering operations [33]. Jardine and van Rijsbergen [21] pointed out that "associations between documents convey information about the relevance of documents to requests." They demonstrated that cluster-based retrieval strategies are as effective as linear strategies, and much more efficient. Thus many clustering methods have been used for information retrieval [21], [7], [17]. The most popular clustering methods are the hierarchical agglomerative clustering (HAC) methods, because their search and construction techniques are more efficient than for most nonhierarchical methods [21].

The following sections define some terminology in cluster analysis, describe the algorithms we used to build the browse hierarchy, and present some samples of the browsing hierarchy obtained for the AIX library.

1) Some Terminology in Cluster Analysis: Classification by cluster analysis has been of long-standing interest in statistics as well as various other fields. It can be traced back to the work of Adanson in 1757 [1], who used numerical clustering for classifying botanic species. Statisticians and taxonomists have widely developed the field since then. Cluster analysis now offers a wide range of techniques for identifying underlying structures in large sets of objects and revealing links between objects or classes of objects. One particular application of classification is the building of libraries.

There is no strict definition of cluster, but it is generally agreed that a cluster is a group of objects whose members are more similar to each other than to the members of any other group. Typically, the goal of cluster analysis is to determine a set of clusters, or a clustering, such that intercluster similarity is low, and intracluster similarity is high. The similarity between objects is evaluated via a numerical measure called a *dissimilarity index* defined as follows.

Definition 1: Let Ω be a set of objects. A **dissimilarity index** δ over Ω^2 is a function from $\Omega \times \Omega$ to R_+ that satisfies the following properties:

$$\text{(i)} \quad \forall o \in \Omega, \quad \delta(o,o) = 0, \tag{5}$$

$$\text{(ii)} \quad \forall (o,o') \in \Omega^2, \delta(o,o') = \delta(o',o). \tag{6}$$

Note that a distance is a dissimilarity index, but that a dissimilarity index does not necessarily satisfy the triangle inequality and therefore is not a distance.

The dissimilarity index between objects is used as the basic criterion to determine clusters. Clustering techniques allow identifying not only clusters, but also relationships among them. The structure of the set of clusters as well as their internal structure vary with the clustering technique. Clustering methods are usually classified[9] according to the structure of the set of clusters produced—e.g., hierarchical, flat, overlapping, etc.—as well as the technique used—e.g., divisive, agglomerative, incremental, etc. As explained previously, hierarchical agglomerative techniques are very convenient for building browse hierarchies. The basic principle that these techniques follow is presented below.

[9] With the recent introduction of conceptual clustering [31], another distinction has been introduced according to the definition of the clusters obtained in extension (i.e., by enumeration of its members) for regular (or numerical) clustering and in intension (i.e., by membership rules) as well as in extension for conceptual clustering.

Hierarchical numerical clustering aims at building hierarchies over a set of objects in which each internal node corresponds to a cluster of objects and each leaf represents an individual object, or more precisely, a singleton cluster. Most hierarchical clustering methods are based upon the same general method, called the Hierarchical Agglomerative Clustering (HAC) method [12], which consists of iteratively gathering objects into clusters until only one cluster remains.

The HAC general method iteratively builds a sequence of partitions or *level clusterings* of Ω; that is, a sequence of disjoint clusters covering the original set of objects Ω. The level clusterings form coarser and coarser partitions by an iterative process, beginning with the level clustering formed by the set of singletons in the power set $\wp(\Omega)$, i.e., $\{\{o_1\}, \{o_2\}, \ldots, \{o_n\}\}$, and ending up with the coarsest partition of Ω, i.e., $\{\Omega\}$. The final output of this clustering process is a particular form of hierarchy called a *dendogram*. The HAC general method can be expressed as follows:

1) **Start** with the subset of $\wp(\Omega)$ formed by singleton elements
2) **Repeat** the following steps iteratively **until** there is only one cluster

   a. **Identify** the two clusters that are the most similar
   b. **Merge** them together into a single cluster.

The HAC method requires a measure of similarity not only over the set of objects, but also over the set of clusters. The dissimilarity index between clusters is usually derived from a user-given dissimilarity index δ between objects. The way of defining Δ has a direct influence on the final form of the hierarchy obtained. Once a dissimilarity index δ between objects is provided, HAC methods differ only by the choice of this measure. The most commonly used HAC methods are the single link and complete link [12]. Many other methods such as the centroid method, Ward's method, etc., define still other dissimilarity indices, but most of them require the dissimilarity index over Ω to be a distance; that is, to satisfy the triangle inequality. The reader should consult [13], [12] for an extensive survey of the HAC methods. The time complexity of the HAC algorithm is at most $O(n^2 \log n)$, where n is the number of objects involved. For some particular definitions of Δ, it can be reduced to $O(n^2)$.

2) Adapting a Clustering Technique for Building a Browse Hierarchy: As explained above, we propose to use a HAC technique to generate a browse hierarchy. In this perspective, we: (i) need to define a measure of similarity between the objects considered, e.g., the documents, and (ii) explain how to make a browse hierarchy out of the dendogram generated by the HAC technique. Let us address these two points.

In information retrieval, numerous measures of similarity between documents, also termed measures of association or coefficients of association, have been defined [40]. The simplest of all is defined as:

$$|X \cap Y| \tag{7}$$

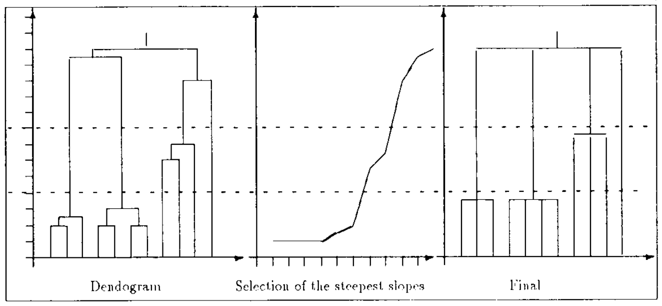


Fig. 2. Principle of selection of level clusterings.

where X and Y are the profiles of two documents. This measure represents the number of common index units. Various other measures [40] have been defined such as:

$$\frac{2|X \cap Y|}{|X| + |Y|} \quad \text{Dice's coefficient} \tag{8}$$

$$\frac{|X \cap Y|}{|X \cup Y|} \quad \text{Jaccard 's coefficient} \tag{9}$$

$$\frac{|X \cap Y|}{|X| \times |Y|} \quad \text{Salton's Cosine coefficient.} \tag{10}$$

They can all be considered as normalized versions of (7), since they are functions of the cardinality of $X, Y, X \cap Y$, or $X \cup Y$.

In our context, we have more information than just the presence or absence of index units in the profile, and therefore we propose to take into account the ρ-values of LA's in the evaluation of the measure of association between documents. For any profile $X = \{(i, \rho)\}$, $p(X)$ is the projection set of X over the set of indices. Then the simplest measure is $|p(X) \cap p(Y)|$; i.e., the number of indices in common in both profiles. In order to take into account the resolving power of indices as well, we define our measure δ for two profiles X and Y, such that $X \neq Y$, as:

$$\delta(X, Y) = \sum_{i \in p(X) \cap p(Y)} (\rho_X(i) \times \rho_Y(i)) \tag{11}$$

where $\rho_X(i)$ is the ρ value of the index i in the profile X, and similarly for Y. Note that δ is a measure of similarity rather than a measure of dissimilarity. Its inverse is a measure of dissimilarity as long as $\delta(X, X)$ is set to a sufficiently large arbitrary value so that its inverse can be considered essentially null.

Given such a measure of similarity between profiles, we define a measure of similarity between clusters according to the single link or complete link techniques, for instance, and then use the hierarchical agglomerative clustering algorithm in order to build a browse hierarchy of software components. Let us note that we also made some experiments in earlier versions of GURU using an incremental conceptual clustering technique [27] for constructing the browse hierarchy. However, despite interesting results, the cost of building and maintaining the hierarchy was prohibitive (exponential time like for most conceptual clustering techniques) when compared to regular clustering techniques, and did not appear to be better in terms of retrieval effectiveness.

```
Let $y_n, \ldots, y_1$ be the merging values of the similarity measure from the bottom to the top.
For $i = n - 1$ to $i = 1$
    $\Delta y_i = y_{i+1} - y_i$
    (evaluate the slope of the connecting segment)
EndFor
Compute $\overline{\Delta y}$ the mean of the $y_i$'s
Compute $\sigma$ the standard deviation of the $y_i$'s around $\overline{\Delta y}$
Let $t(k) = \overline{\Delta y} + k\sigma$
(where $t(k)$ corresponds to a threshold defined by $k \geq 0$)
For $i = n - 1$ to $i = 1$
    If $\Delta y_i > t(k)$
        Select level clustering $i$
    EndIf
EndFor
```

Fig. 3. Selection of level clusterings.

All the HAC techniques build a binary hierarchy. Not all levels of the hierarchy are equally significant; therefore the usual approach is to select manually the most significant level clusterings, this task being usually performed by a data analyst. The following proposes a method for automatically identifying the most useful level clusterings and thus producing a not-necessarily binary hierarchy.

This method of selection is based on the following principle. Each level clustering in the dendogram corresponds to the merging of two clusters in the previous level clustering and therefore to a particular value of the similarity measure. If we label the dendogram with these values, $y_n, \ldots, y_1$, n being the number of objects, from the bottom to the top of the hierarchy, it can easily be shown that the y_i's are (nonstrictly) monotonic (increasing for dissimilarity measures and decreasing for similarity measures) for the single and complete link-clustering methods. We propose to select those levels which correspond to the gap in the distribution of y_i's by (i) plotting the segment connecting the pairs y_{i+1}, y_i from $i = n-1$ to $i = 1$, and (ii) keeping the levels which correspond to the steepest slopes. This represents the intuitive method that a data analyst would apply. Fig. 2 gives an intuitive presentation of the method via an example, whereas Fig. 3 gives the formal algorithm. The time complexity of the latter is linear in the number of objects.

C. Some Examples

Portions of the browse hierarchy built from the AIX doc-

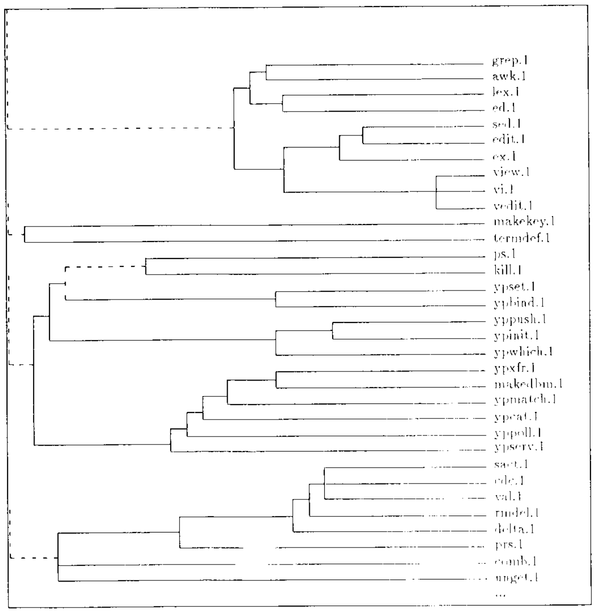


Fig. 4. Portion of AIX hierarchy (single link, $k = 0.5$) .

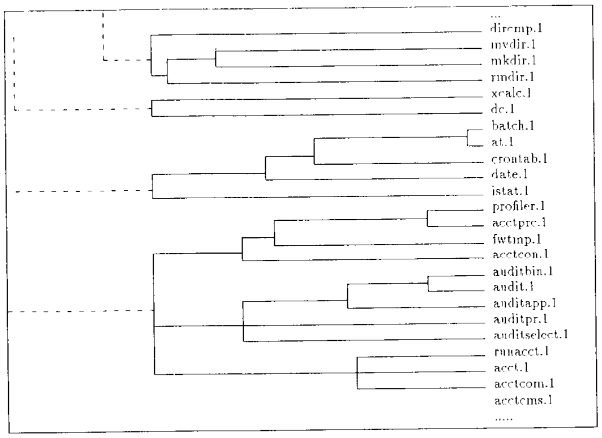


Fig. 5. Portion of AIX hierarchy (single link, $k = 0.5$) .

umentation are shown in Figs. 4 and 5. In Fig. 4, some interesting clusters are isolated. Thus in the figure we have a cluster that gathers commands related to the manipulation of regular expressions, and a cluster that gathers editors. These two clusters are also part of the same supercluster, mainly because these editors permit the manipulating of regular expressions. Then there are two outliers which could not be included in a cluster: `makekey` and `termdef`. Then a small cluster groups `ps` and `kill`, which are strongly related since they give information about processes or handle them. Finally, there are two big clusters, one for yellow pages commands and another for SCCS routines. The clustering is not always of such good quality either because of the nature of the documentation or the principle of clustering itself. For instance, as can be seen in Fig. 5, the commands `xcalc` and `dc`, which are calculators, belong to the same cluster, but `bc` has been forgotten in this cluster. This is due to the fact that the manual page of `bc` does not refer to the concept of calculator at all, but defines `bc` as an interpreter for an arithmetic language. The real problem with clustering is illustrated with the third cluster in this figure, which gathers `batch`, `at`, `crontab`, `date` and `istat`. This cluster has been formed because all these commands are related to the notion of date or time; unfortunately, this is not the main functionality of all of these commands and therefore this cluster is somehow misleading. Let us note, however, that the lower level cluster including `at` and `batch` is a good one.

The hierarchy thus generated is used as an aid to browse when nothing relevant has been retrieved via linear retrieval or in order to increase recall, since there is no way to be sure that all the relevant components have been retrieved at the linear retrieval stage. It can also be used as the basic repository to be searched during retrieval, but we prefer to use the traditional linear-retrieval technique instead, because it is clearly more trustable considering the problems described above.

By nature, this indexing technique suffers from noise, since it is based on only statistical observations. Noisy indices involve generally misspelled or unmeaningful strings of characters that are mixed with natural language (for describing instructions, for instance), or "side-concepts" such as the time, day, and month in the example cited above. This noise cannot be avoided when dealing with free-style text.

Fortunately, these noisy LA's do not cause real trouble at the linear retrieval stage, since there is a very low probability that the user would use unmeaningful character strings in her/his queries. So noisy LA's are part of the profiles of components but rarely lead to the selection of the considered component. On the other hand, noisy LA's might induce the formation of poor quality clusters, but generally only higher levels of the hierarchy are affected, since "side concepts" are not given much weight when evaluating similarity. Section V-C explains how this browsing hierarchy is used at the retrieval stage.

V. The Retrieval Stage

The previous sections explained how libraries of reusable components are assembled. We also need to be able to retrieve the components which match the requirements when at least one exists, or to assist in the selection of the closest components via a browsing facility.

The usual scenario when retrieving a component is the following:

1) **Query specification:** The user expresses a query according to the authorized vocabulary
2) **Linear retrieval:** A search locates the candidate components and the candidates are ranked according to their degree of match with the query
3) **Browsing:** Cluster-based retrieval is initiated when no adequate components have been found by the linear retrieval.

The following explains how these three stages are supported in our approach.

```
Get natural-language query $q$ from user
Index $q$ and produce its profile $\mathrm{Prof}(q) = \{(i, \rho_q(i))\}$
For each query index, $(i, \rho_q(i)) \in \mathrm{Prof}(q)$
    $C(i) \leftarrow \{(d, \rho_d(i))\}$
    (i.e., retrieve the information associated to $i$ in the inverted file index)
EndFor
$C \leftarrow \cup\{C(i)\}_{(i,\rho) \in \mathrm{Prof}(q)}$
For each $d$ such that $(d, \rho_d(i))$ in $C$
    Evaluate the similarity between the query and $d$ as
        $\delta(q, d) = \sum_i \rho_d(i) \times \rho_q(i)$
        ($\delta$ is the similarity measure defined in (11))
EndFor
Rank components in order of decreasing similarity.
```

Fig. 6. Linear retrieval technique.

```
Processing query:
How can I locate a regular expression in a file
Lemmatizing sentence...
Searching...
regex.3  220.21
regexp.3  220.21
awk.1    77.32
grep.1   77.32
find.1   33.88
egrep.1  28.77
regcmp.3 28.77
dosfirst.3 22.38
dosnext.3 22.38
```

Fig. 7. Example of linear retrieval.

A. Query Specification

Using uncontrolled-vocabulary indexing, as we do, presents clear advantages at the query specification stage. Indeed, a minimum of constraint is put on the user as he/she expresses his/her query. The user does not have to learn a specific index language or understand the organization of the library. He/she can express his/her query in natural language, and then the indexing component is applied in order to translate the query into attributes understandable by the system. Exactly the same technique is used for extracting LA's from natural-language queries as from natural-language documentation. This provides a very convenient and user-friendly interface between the user and library system, because the user is not constrained by any rigid formalism.

The queries can be expressed in free-style natural language. However, the user must be aware of the fact that queries are not really interpreted, but are rather considered as a description of the functionality of the desired component. For instance, the user could express queries of the form, "how can I do such and such," since only the "such and such" would be considered for indexing, the rest being either closed-class words or words with a low quantity of information. Formulating a query which necessitates some understanding, such as a query including negations such as "but not," would only lead to wrong interpretation. Let us note that it would be possible at this point to allow some simple interpretation of the queries by allowing, for instance, the usual Boolean connectors ("and", "or", "but not"). This would clearly boost the performance of the library system. However, since our point here is to show how far we can go without understanding either the queries or documents, we do not discuss these possible enhancements.

B. Linear Retrieval

In order to retrieve the best candidates for a given query we apply the usual IR method, which consists of considering the query as a document and retrieving the components in the repository whose profile is the most similar to the profile of the query. A possible measure of similarity is the δ measure defined in (11). The most similar components are then returned to the user, ranked in order of decreasing similarity with the query. The linear retrieval technique is presented in Fig. 6.

In case of low recall—that is, if the user is not satisfied with the retrieved candidates—a more fuzzy search can be performed that also considers partially matching LA's. In that case, only LA's which partially match a query LA (i.e., have one word in common) are considered. This significantly increases the recall, but as a trade-off drastically decreases the precision. It should therefore be used only when the user considers that nothing relevant has been retrieved with the initial query. An example of linear retrieval is given in Fig. 7.

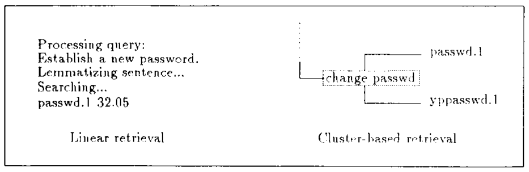
In Fig. 7 the candidates are ranked in order of decreasing similarity with the query ("How can I locate regular expressions in a file"). Therefore the top candidates usually answer the query the best. In the example shown in Fig. 7, all the candidates retrieved deal more or less strongly with regular expressions. Even the two last candidates, `dosfirst` and `dosnext`, do not answer the query but are very slightly related, since they allow locating DOS files which match a pattern.

C. Browsing, Cluster-Based Retrieval

The retrieval stage in classical library management systems is often limited to locating a set of components exactly matching the user's query or, when such components do not exist, related components. Library systems do not usually provide any further assistance.

In our approach, the user may communicate interactively with the system in order to direct the browsing when he/she is not satisfied with the first retrieval yielded. The linear search retrieves the most related candidates, and then the browsing process begins.

Typically, the user starts from one of the candidates retrieved by the linear search and explores the hierarchy bottom-up. Consider the browse hierarchy given in Fig. 4 and suppose that a user gives a query asking about ways "to identify a process." If the first candidate retrieved at the retrieval search is `kill`, then the user can access the browse hierarchy and explore the clusters that include `kill` in order to determine which components are strongly related. In our example, the user will find `ps` as the most related component, which is clearly a better candidate for this given query than the one retrieved by the linear search. Another example is illustrated in Fig. 8. The two relevant candidates in AIX for the query "establish a new password" are `passwd` and `yppasswd`.



```
Processing query:
Establish a new password.
Lemmatizing sentence...
Searching...
passwd.1 32.05
```

Fig. 8. Browsing in the hierarchy.

However, the linear retrieval retrieves only passwd simply because the query had no intersection with the profile of yppasswd. At this point the user could reformulate the query, but he/she might not be aware that he/she has missed some relevant candidates. Using the browse hierarchy is therefore more convenient in order to check if some unexpected candidates have been missed. In the example, both passwd and yppasswd are strongly related: their profiles share the LA *(change passwd)*[10] and therefore belong to the same low-level cluster in the browsing hierarchy. Browsing in the hierarchy from passwd allows the user to retrieve the other relevant candidate. These two examples show how a browse hierarchy can help improve the finding of possible candidates that could be missed via linear retrieval.

At any point the user can consult the profile of a component in order to have more information about its functionality. Fast access to profiles is achieved via the profile repository. The user can also provide, at any stage, further information in order to get a finer retrieval. By browsing, he/she gets more information about components and learns how to provide discriminating queries.

VI. Empirical Results

The approach described in the previous sections has been embodied in a tool, GURU, which has been fully implemented, in C, under AIX on an RS/6000. The system has reached a satisfactory first stage and the implemented version yields quality results.

We have tested our system on the entire AIX documentation available to us, which describes approximately 1100 AIX components. When building the index repository, we therefore processed the entire documentation which forms a corpus of more than 800 000 words, and we identified 18 000 LA's for the 1100 profiles.

In order to evaluate GURU's performance, we used the following criteria:

- **User effort.** This consists of all the effort which must be expended by the user in order to use the library system. It is very difficult to formally measure user effort. However, thanks to the uncontrolled vocabulary approach which we applied, we believe that the effort which must be invested for using GURU is minimal. Queries can be formulated in natural language, and therefore the user is not required to learn any index language and formalism
- **Maintenance effort.** This consists of all the effort which is necessary to keep the system working and up to date. This effort includes, in particular, indexing new components and adding them to the library. The maintenance stage is highly facilitated in GURU. The indexing is performed automatically and insertion of new components can be done incrementally. Kaplan and Maarek [22] have proposed several algorithms for incrementally updating a repository of LA-based indices when inserting, deleting, or modifying components
- **Efficiency.** This refers to the average interval between the time a query is issued and the time an answer is given. Efficiency becomes an issue only if a retrieval takes so long that users start to complain. Our experience with the system shows that efficiency is not an issue, as the response time is reasonable. Profiling the execution of the query program showed that the time to perform the query was dominated by the time to map the repository file into the address space of the query program. The lookup operations and the printing of the LA-file name pairs consumed almost no time in comparison. Test queries involving from 5–15 LA's each took approximately 2.5 s on an RT, and 0.15 s on an IBM RISC System/6000. The better performance of the latter is partly due to its more efficient implementation of file mapping
- **Retrieval effectiveness.** This is clearly the most important performance criterion. It refers to the system's ability to provide information services as needed by the user.

[10]Note that "passwd" here is a proper name and different from the noun "password" mentioned in the query.

The next section focuses on evaluating the retrieval effectiveness of GURU.

Measuring Retrieval Effectiveness

1) Recall and Precision: The most widely used measures for evaluating retrieval effectiveness are *recall* and *precision* [34]. Recall is defined as the proportion of *relevant* material; i.e., it measures how well the considered system retrieves *all* the relevant components. Precision is defined as the proportion of retrieved material which is relevant; i.e., it measures how well the system retrieves *only* the relevant components. Recall can also be interpreted as the probability that a relevant component will be retrieved, and precision as the probability that a retrieved component will be relevant [5].

Recall and precision can be defined more formally as follows: Let C be the whole collection of components forming the library. For each query, C can be partitioned into two disjoint sets, R, the set of relevant material, and $\bar{R}$, the set of irrelevant material. Given the query, the system retrieves a set of components c that can also be partitioned into relevant and irrelevant material, respectively, r and $\bar{r}$. Recall and precision are defined as:

$$\text{recall} = \frac{r}{R} \tag{12}$$

$$\text{precision} = \frac{r}{c}. \tag{13}$$

Recall and precision measurement require the ability to distinguish between relevant and irrelevant material. Relevance judgments are always debatable, and it is a very tedious task to produce test collections with adequate relevance judgments. To our knowledge, no test collection for software documentation is available. Therefore we produced such a test collection—i.e., a set of queries and the associated set of relevant material—for the AIX documentation. The test collection is described in the next section.

2) Experiments and Comparison: This section describes the experiments which allowed us to evaluate the retrieval effectiveness of GURU As a basis for comparison, we have considered INFOEXPLORER, which is an IBM RISC System/6000 CD-Rom Hypertext Information Base Library [19]. INFOEXPLORER is a recent hypertext system that gives access to the documentation for AIX and to associated programs. INFOEXPLORER provides not only hypertext links between pieces of the AIX documentation, but also search and retrieval facilities. Queries can be expressed as single-word search or multiple-word compound search with no control of vocabulary. The compound search, which is the most elaborated, allows the user to express a query as a word pattern formed of single words related by three possible connectors, "and", "or", and "but not". Moreover, the user can restrict the search. He/she can give constraints specifying if the pattern words must appear within the same article or within the same paragraph, the proximity of these words within a paragraph, and the search fields and search categories.

When given such a query, INFOEXPLORER returns a list of candidates that exactly fit the query, ranked according to the frequency of the pattern in the considered document. No profile is built for the documents examined: all words appearing in the text are considered during the search. Therefore, INFOEXPLORER can be expected to have a much higher recall but lower precision than GURU. We do not need to also compare efficiency; i.e., retrieval speed. GURU is, independently of implementation, much faster than INFOEXPLORER, since it does not explore the entire textual database but a much smaller repository formed by the profiles.

INFOEXPLORER is thus a commercial IR tool which represents a good reference for comparison purposes, since it is specifically for AIX. Also, INFOEXPLORER encodes a great deal of manually provided information about the structure of the documentation. The system has to know about paragraphs, titles, etc., and thus has been much more expensive to build than GURU. Providing this structural information to our system would greatly enhance its performance, but our point here is to show that even without such information, our system can perform nicely thanks to its indexing scheme.

GURU and INFOEXPLORER were compared for retrieval effectiveness. In order to claim this test to be valid, we must fulfill the usual test procedure requirements [34]. These requirements are for:

1) the queries to be used for test purposes must be user search requests actually submitted and processed by both systems,
2) the test collection must consist of documents originally included in the library, chosen in such a way that any advance knowledge concerning the retrievability of any given component by either system is effectively ignored,
3) the number of components considered to be retrieved by the two systems must be subject to the same cutoff.

To fulfill the first requirement, we conducted a survey among graduate students in the Department of Computer Science at Columbia University in November 1988. This survey provided us with a collection of typical queries on UNIX-like systems, as formulated by UNIX users ranging from naive users to expert programmers. A typical query was expressed as a natural-language sentence with an average of 3.7 open class words per query describing a desired functionality. This kind of query could directly be fed to GURU but not to INFOEXPLORER, since the latter's compound search facility accepts only Boolean queries. Therefore feeding the queries to INFOEXPLORER required some supplementary effort—first choosing the right connectors between open-class words extracted from the queries, and possibly dropping some words when the recall was too low. In our interaction with the compound search facility we had to refine and retry the query formulation several times. We kept only the best result for comparison purposes, since we wanted to compare the tools' indexing schemes rather than their querying facilities. GURU's querying facility requires less user effort than INFOEXPLORER's, but the latter's could be greatly improved if it did not require perfect matches between the Boolean query and the candidates, using a similarity measure between candidate and query, for instance. The average number of open-class words used for questioning INFOEXPLORER was 3.

As far as the second requirement is concerned, the collection considered for test has been the entire AIX library. We consulted with several AIX experts at IBM in order to determine for each query the set of existing relevant components in the AIX library so as to be able to evaluate the recall and precision. As our test collection was composed of about 1100 components, we selected 30 queries from among all the queries provided by our survey. This ratio corresponds to the same number-of-queries per number-of-documents ratio as the one which has been used in standard test sets such as MED (collection of medical abstracts, 30 queries for 1033 documents) or CISI[11] (information science abstracts, 35 queries for 1460 information abstracts).

As far as the third requirement is concerned, since both systems ranked the retrieved candidates, we were able to compare recall and precision at the same ranks.

The comparison was performed by measuring, for both systems, precision at several levels of recall. We followed the usual procedure [40], [34], which consists of:

1) Plotting precision-recall curves for each test query with each plot corresponding to a given cutoff value
2) Extrapolating these curves so as to obtain precision values for recall values which were not effectively achieved
3) Deriving from the curves computed in stage (2) the average precision values at fixed recall intervals so as to obtain a single average precision recall curve for the system considered.

We have built such curves for both GURU and INFOEXPLORER and plotted them on the same axes (See Fig. 9). The best

[11]These test sets have been used for evaluating several IR systems such as LSI [9].

Recall	GURU precision	INFO precision	Improvement
0.1	0.85	0.7	0.15
0.3	0.84	0.68	0.15
0.5	0.76	0.56	0.20
0.7	0.58	0.4	0.18
0.9	0.52	0.39	0.13

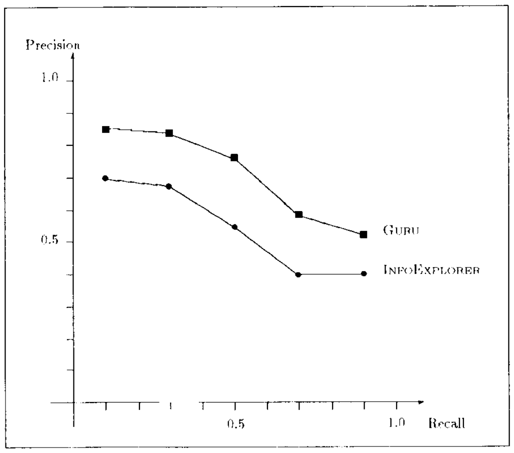


Fig. 9. Precision-recall curves (means across queries).

performance is reached by the system whose curve is closest to the area where both precision and recall are maximized, the upper right corner of the graph. As was mentioned, because of the indexing scheme of both systems we could expect that INFOEXPLORER would achieve a lower precision but higher recall than GURU. It turned out that the maximum recall, all ranks included, achieved by both systems was approximately the same, around 88% on the average, but from the graph presented in Fig. 9, it is clear that GURU had 15%, on average better precision than INFOEXPLORER.

These results show that for the sample tested, GURU achieves higher precision than INFOEXPLORER without losing in recall. For this sample, the recall rate is increased when we make use of the GURU browse facility. For instance, in several cases, some related components were not retrieved during linear retrieval, but only during browsing.

The results of this evaluation should not be seen as final definitive results, but only as an indication of what can be expected from the GURU system. Until more test collections specifically designed for software documentation become available, it is not possible to produce statistically significant results. Producing large-scale collections requires a great deal of effort and is out of the scope of this work, but we hope that our work, as well as the work of others, will motivate this effort. In the meantime, however, our results are very promising.

VII. CONCLUSION

We have presented a method for automatically constructing software libraries from a collection of documented but unindexed software components. We discussed the advantages of using natural-language documentation as opposed to source code, assuming that any documentation is available, as a source of functional information. We then described a new free-text indexing scheme for automatically producing document profiles based upon a richer unit than single terms, the lexical affinity. All associated software components could then be classified, stored, compared, and retrieved via linear or cluster-based techniques according to these indices.

These methods and schemes are embodied in a new tool which has been implemented and evaluated for retrieval effectiveness. The evaluation compared GURU with the INFOEXPLORER hypertext library, built specifically to help find software components in the AIX system. The average recall-precision curves of both tools were computed. The results of this test indicate that GURU 's performance was better than INFOEXPLORER. This result is very encouraging, since INFOEXPLORER was much more expensive to build and specifically tailored to the AIX library.

The major contribution of this work consists of bringing classical and new information retrieval techniques to bear in software reuse. This involved:

1) Designing a new indexing scheme based on high information content lexical affinities
2) Adapting classical numerical cluster analysis techniques for assembling software components into browse hierarchies
3) Designing retrieval mechanisms specifically adapted to the LA-based indexing scheme so as to provide a complete storage and retrieval framework.

Finally, the evaluation we have performed seems to indicate that Salton's statement about the limitation of the "phrase generation" approach in indexing (see Section III-A) is overly pessimistic, and that significant improvements over single-term techniques can be achieved at relatively low cost.

ACKNOWLEDGMENT

The authors would like to thank M. Kennedy, who helped in the design and implementation of GURU's retrieval component.

REFERENCES

[1] M. Adanson,*Histoire Naturelle du Sénégal. Coquillages. Avec la relation abrégée d'un voyage fait en ce pays, pendant les années 1749,50,51,52 et 53.* Paris: Bauche, 1757.
[2] B. P. Allen and S. D. Lee, "A knowledge-based environment for the development of software parts composition systems," in *Proc. 11th ICSE* (Pittsburgh, PA), May 1989, pp. 104–112.
[3] S. P. Arnold and S. L. Stepoway, "The reuse system: Cataloging and retrieval of reusable software," in *Software Reuse: Emerging Technology*, W. Tracz, Ed. Los Alamitos, CA: IEEE Computer Soc., 1987, pp. 138–141.
[4] R. Ash, *Information Theory.* New York: Wiley–Interscience, 1965.
[5] D. C. Blair and M. E. Maron, "An evaluation of retrieval effectiveness for a full-text document retrieval system," *Commun. ACM*, vol. 28, no. 3, pp. 289–299, Mar. 1985.
[6] B. A. Burton, R. Wienk Aragon, S. A. Bailey, K. D. Koelher, and L. A. Mayes, "The reusable software library," in *Software Reuse: Emerging Technology*, W. Tracz, Ed. Los Alamitos, CA: IEEE Computer Soc., 1987, pp. 129–137.
[7] F. Can and E. A. Ozkarahan, "A clustering scheme," in *Proc. SIGIR'83* (Bethesda, MD), 1983, pp. 115–121.
[8] F. de Saussure, *Cours de Linguistique Générale, Quatrième Edition.* Paris: Librairie Payot, 1949.

[9] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman," Indexing by latent semantic analysis," *J. Amer. Soc. Inform. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.

[10] P. Devanbu, "Re-use of software knowledge: A progress report," presented at the 3rd Ann. Workshop: Methods and Tools for Reuse, Syracuse, NY, June 1990.

[11] P. Devanbu, P. G. Selfridge, B. W. Ballard, and R. J. Brachman, "A knowledge-based software information system," in *Proc. IJCAI'89* (Detroit, MI), Aug. 1989, pp. 110–115.

[12] E. Diday, J. Lemaire, and F. Testu, *Eléments d'Analyse des Données*. Paris: Dunod, 1982.

[13] B. Everitt, *Cluster Analysis*. New York: Halsted, 1980.

[14] W. B. Frakes and P. B. Gandel, "Classification, storage and retrieval of reusable components," in *Proc. SIGIR'89* (Cambridge, MA), June 1989, N. J. Belkin and C. J. van Rijsbergen, Eds., pp. 251–254.

[15] W. B. Frakes and P. B. Gandel, "Representing reusable software," *Inform. Software Technol.*, Nov. 1990.

[16] W. B. Frakes and B. A. Nejmeh, "Software reuse through information retrieval," in *Proc. 20thAnn. HICSS* (Kona, HI), Jan. 1987, pp. 530–535.

[17] A. Griffiths, L. A. Robinson, and P. Willett, "Hierarchical agglomerative clustering methods for automatic document classification," *J. Documentation*, vol. 40, no. 3, pp. 175–205, Sept. 1984.

[18] W. Harrison, "A program development environment for programming by refinement and reuse," in *Proc. 19th HICSS* (Kona, HI), 1986, pp. 459–469.

[19] *IBM AIX Version 3 for RISC System/6000. Commands Reference.* Yorktown Heights, NY: IBM, 1990.

[20] T. Ichikawa and M. Hirakawa, "Ares: A relational database with the capability of performing flexible interpretation of queries," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 624–634, May 1986.

[21] N. Jardine and C. J. van Rijsbergen, "The use of hierarchic clustering in information retrieval,"*Inform. Storage and Retrieval*, vol. 7, no. 5, pp. 217–240, Dec. 1971.

[22] S. M. Kaplan and Y. S. Maarek, "Incremental maintenance of semantic links in dynamically changing hypertext systems," *Interacting with Computers*, vol. 2, no. 3, Dec. 1990.

[23] P. H. Klingbiel, "Machine-aided indexing of technical literature," *Inform. Storage and Retrieval*, vol. 9, pp. 79–84, 1973.

[24] G. N. Lance and W. T. Williams, "A general theory of classificatory sorting strategies," *Computer J.*, vol. 9, pp. 373–380, 1967.

[25] M. Luhn, "The automatic creation of literature abstracts," *IBM J. Res. Develop.*, vol. 2, no. 2, pp. 159–165, Apr. 1958.

[26] Y. S. Maarek, "Using structural information for managing very large software systems," Ph.D. thesis, Technion, Israel Instit.Technol., Haifa, Israel, Jan. 1989.

[27] Y. S. Maarek, "An incremental conceptual clustering algorithm with input-ordering bias correction, in *Advances in Artificial Intelligence, Natural Language and Knowledge Base Systems*, M. C. Golumbic, Ed. New York: Springer-Verlag, 1990.

[28] Y. S. Maarek and G. E. Kaiser, "On the use of conceptual clustering for classifying reusable ada code," in *Proc. Ada Letters, Using Ada: ACM SIGAda Int. Conf.* (Boston, MA), Dec. 1987, pp. 208–215.

[29] Y. S. Maarek and F. A. Smadja, "Full text indexing based on lexical relations, an application: Software libraries," in *Proc. SIGIR'89* (Cambridge, MA), June 1989, N. J. Belkin and C. J. van Rijsbergen, Eds., pp. 198–206.

[30] W. J. R. Martin, B. P. F. Al, and P. J. G. van Sterkenburg, "On the processing of a text corpus: From textual data to lexicographic information," in *Lexicographiy: Principles and Practice* (Applied Language Studies Series), R. R. K. Hartmann, Ed. London: Academic, 1983.

[31] R. Michalski and R. Stepp, "Automated constructions of classifications: Conceptual clustering versus numerical taxonomy," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-5, pp. 396–409, July 1983.

[32] R. Prieto Diaz and P. Freeman, "Classifying software for reusability,"*IEEE Software*, vol. 4, pp. 6–16, Jan. 1987.

[33] G. Salton, *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.* Reading, MA: Addison-Wesley, 1989.

[34] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval* (Computer Series). New York: McGraw-Hill, 1983.

[35] G. Salton and M. Smith, "On the application of syntactic methodologies in automatic text analysis," in *Proc. SIGIR'89* (Cambridge, MA), June 1989, pp. 137–150.

[36] R. W. Schwanke, R. Z. Altucher, and M. A. Platoff, "Discovering, visualizing and controllling software structure," in *Proc. 5th Int. Workshop on Software Specifications and Design* (Pittsburgh, PA), May 1989, pp. 147–150.

[37] F. A. Smadja, "Lexical co-occurrence: The missing link,"*J. Assoc. Literary and Linguistic Computing*, vol. 4, no. 3, 1989.

[38] K. Sparck Jones and J. I. Tait, "Automatic search variant generation," *J. Documentation*, vol. 40, no. 1, pp. 50–66, Mar. 1984.

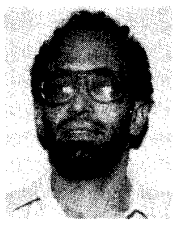
[39] W. F. Tichy, R. L. Adams, and L. Holter, "NLH/E: A natural-language help system," in*Proc. 11th ICSE* (Pittsburgh, PA), May 1989, pp. 364–374.

[40] C. J. van Rijsbergen, *Information Retrieval*, 2nd ed. Stoneham, MA: Butterworths, 1979.

[41] M. Wood and I. Sommerville, "An information retrieval system for software components," *SIGIR Forum*, vol. 22, nos. 3/4, pp. 11–25, Spring/Summer 1988.

Yoëlle S. Maarek graduated from the "Ecole Nationale des Ponts et Chaussées," Paris, France, in 1985. She completed the D.E.A. (graduate degree) in computer science from Paris VI University in 1985, and received the Doctor of Science degree from the Technion, Israel Institute of Technology, Haifa, in 1989.

She has been a Research Staff Member in the Software Environments Department at the IBM T. J. Watson Research Center, Yorktown Heights, NY, since 1989, where her research interests include programming environments, software reuse, and information retrieval.



Daniel M. Berry received the Ph.D. degree in computer science from Brown University in 1973.

He was on the faculty of the Computer Science Department at the University of California, Los Angeles, from 1972 to 1987. Since 1987 he has been a Professor in the Faculty of Computer Science at the Technion, Israel Institute of Technology, Haifa. He is currently on leave from Technion at the Software Engineering Institute, Pittsburgh, PA. He has consulted with the Verification Group at Unisys, Culver City, CA, since 1980. His areas of research interest are in software engineering, with emphases on requirements elicitation and programming environments and in multilingual word-processing.

Dr. Berry is a member of the Association for Computing Machinery and the IEEE Computer Society.

Gail E. Kaiser received the Sc.B. degree from the Massachusetts Institute of Technology, Cambridge, and the M.S. and Ph.D. degrees from Carnegie Mellon University, Pittsburgh, PA.

She is an Associate Professor of Computer Science at Columbia University, New York City, and has published over 50 papers in a wide range of software areas, including software development environments, testing and debugging tools, extended transaction models, reusability, application of artificial intelligence technology to software engineering, object-oriented languages and databases, and parallel and distributed systems.

Dr. Kaiser was selected as an NSF Presidential Young Investigator in Software Engineering in 1988, and received a Digital Equipment Corporation Incentives for Excellence Award in 1986.