A Software Reuse System for C Codes

Le van Huu*

TR-93-067

December 2, 1993

Abstract

This paper presents PRASSY, a hypertext system for the storage and retrieval of procedure source codes, on the basis of the semantics of their comments. The objective of the system is to provide the program developer with the possibility of retrieving and reusing the source code of C subroutines that have been previously built by his colleagues or that are already present in the system. The approach adopted by PRASSY is the analysis of the source code comments and of the specification documents written in natural language, in order to extract indexing information. Such information is organized in a hypertext structure and the browsing mechanism is used by the user to select reusable software components. The system provides a way for measuring the semantic similarity between the user requirements and the candidate node to be selected.

The paper describes the system's architecture and functionalities. Some examples of the user interface and the browsing mechanisms are reported. Finally, it describes the algorithm proposed by Aragon-Ramirez and Paice and adopted by PRASSY for defining the semantic similarity among phrases expressed in natural language.

KEYWORDS: hypertext, software reuse, semantic phrases similarity

*Dipartimento di Scienze dell'Informazione, Universita' degli Studi di Milano, Via Comelico, 39, I-20135 Milano, Italy, Phone: +39-2-55006-356, Fax: +39-2-55006-253, Email: levan@imiucca.csi.unimi.it

1 Introduction.

As Bassett outlined in [Bassett87], it is ironic that software - the very engine of our high-tech, automated society - is such a low-tech cottage industry. In other words, the software designers often have to reinvent existing solutions or proceed in an expensive way for resolving their problems. For this reason, for several years researchers have been paid their attention to the possibility of reusing existing software for new projects. Software reuse, if well applied, could reduce the efforts of people involved in the software development process and determinate a high improvement in productivity (about 35-85 percent [Jones84]). As outlined by Adams in [Adams93], the cost for implementing the successive release of the software after the initial delivery is around 80% of the total cost.

The term software reuse, or software reusability, is generally intended as the source code adaptation and interchange between software projects. But it could also be applied to every activity of the software development process, from the requirement analysis to the project documentation, including knowledge dissemination.

Reusing software needs an adequate information organization in a reuse library. This library can contain different information, such as fragments of the specification document, design components, pieces of the source code, according to the specific phase of the software life cycle it refers to. The works for an user who intends to utilize the library content could be summarized in: selection of the candidate component, adaptation of the selected component to the new requirements, integration of the modified component to the current project[Redwine89]. Moreover, if the user believes that some of his project components can be useful to other software developers, a cataloguing work for increasing the library information is necessary. There are several proposals for supporting some or all the above activities, such as the Meld declarative language from the Columbia University and the Carnegie Mellon University [Kaiser87], the incremental implementation model proposed by the Software Productivity Consortium [Prieto-Diaz91], and the building blocks approach for systems programming from IBM [Lenz87]. An exhausted list of other works can be found in [Tracz90].

Krueger in [Krueger92] partitions the approaches to software reuse into eight categories: high-level languages, design and code scavenging, source code components, software schemas, application generators, very high-level languages, transformational systems, and software architectures. We shortly describe those categories that concern our work. The *design* and *code scavenging* refers to the activities of the programmers to select in an informal way fragments of existing software for the new project. In fact, the access method is only in the mind of the programmers. The *source code component reuse* category is similar to the design and code scavenging category, but it uses systematic techniques to select reusable components, for example, organizing them in appropriate libraries.

The *software schema* category emphasizes the reuse of abstract algorithms and data structures rather than source code. An example of this category comes from PARIS [Katz92], a system implemented in LISP for reusing algorithms used in distributed programming. It consists of a library of partially interpreted schemas, that form a program skeleton where some parts remain abstract or undefined. The user query contains specification regarding an entity list, applicability conditions and result assertions. The abstract parts of the selected candidate schema that satisfy the user requirements can be replaced by concrete entities. The replacement can be done manually or automatically.

Our proposal, presented in this paper, belongs to the source code component category and refers specifically to the coding phase of the software life cycle. The objective of the approach is to provide the program developer with the possibility of retrieving and reusing the source code of subroutines that have been previously built by his colleagues or that are already present in the system. The following reasons justify our work. Although the use of subprogram libraries for common functions can reduce the programming effort, it is also true that for accessing these libraries, the programmer has to know about their existence. Normally, one can obtain this information by consulting voluminous manuals or by means of his own programming experience. Moreover, the use of each library function may vary in a limited way, according to the number of its formal parameters. New approaches, such as Object-Oriented Programming paradigm, allow a more flexible way of reusing existing data and procedures (methods) by means of classes and object inheritance concepts. Again, the programmer needs to know about what he can inherit. On the other hand, it is not difficult to obtain the source code of applications because several organizations distribute it freely or at a low price. Good examples come from the early version of the UNIX [1] operating system, the X Window system [2], the software distributed by the Free Software Foundation and programs that one can get from the networks using ftp commands. Finally, most programmers use the code scavenging method for reusing software. Providing the user with a systematic way for retrieve software components is an important goal.

Locating useful subroutines in program source codes, on the basis of their functionality, requires an adequate reuse system. To do this, first we must choose the appropriate technique to create the data base of the reusable information. As mentioned in [Maarek91], there are two main approaches: the information retrieval technique (IR), based on the free-text indexing and the Artificial Intelligence technique based on the knowledge on the reused components. The differences between these approaches are well outlined in the cited article.

Another issue is the source of information from which we can extract indexes to build the classification scheme of the reusable components. One approach considers the extension of the programming language to describe the semantic of the component to be classified. It is adopted by Anna (Annotated Ada), a collection of formal annotations for Ada statements[Luckham84]. These annotations consist of predicates that describe constraints on Ada constructs. We can find the similar idea in Sather [Omohundro93], the object-oriented-language developed by the International Computer Science Institute (Berkeley, California). It consists of the construct *assert*, which includes a key for a controlled program compilation and a boolean expression. If the key is specified to the compiler, then the boolean expression is evaluated. Even if the nature of the construct *assert* is for the program debuging use, it could be extended to describe in an appropriate way the semantic of a program component. On the other hand, in the case of specific domain, e.g., in a mathematical environment, the keywords classification approach, with an informal description in natural language could be sufficient[Krueger92].

The strategy adopted by our system is the analysis of the source code comments and of the specification documents written in natural language, in order to extract indexing information. There are several systems based on this idea, such as RSL[Burton87] which

[1] UNIX is a registered trademark of Unix Systems Laboratories.

[2] X Window System is a trademark of the Massachusetts Institute of Technology.

catalogues the software by attributes (e.g., keywords, authors ...) extracted from comments. CATALOG[Frakes87] recognizes indexes contained in C programs headers, coded in natural language; GURU[Maarek91] analyzes phrases contained in the software documentation and the source codes. Moreover, Prieto-Diaz and Freeman [Prieto-Diaz89] describe a system based on faceted schema and analyze several program descriptions and source listings to determine relevant attributes that one can use to represent programs.

Finally, we must answer the crucial question for any information retrieval application: how can we classify the library components?

The classification schema is central to code accessibility, as asserted by Prieto-Diaz and Freeman in [Prieto-Diaz89]. For their system, they propose the use of faceted schema. Instead of classifying objects in a hierarchical way, they group the objects into different classes, in according to the syntactic or attribute-level relationships. The classifier adopts the synthesis process, analyzing the attribute of objects and arrange them into compound classes called *facets*. The elements of a class are called *terms*. The faceted schema proposed by Prieto-Diaz and Freeman is applied to a collection of very large reusable components with large groups of similar components. The system introduces the conceptual graph to provide a metric to measure the conceptual distance between terms in each facet. In such a graph, the leaves represent facet terms and nodes represent supertypes of terms. Every edge is qualified by a weight value, assigned by the user.

Fuzzy set logic, as proposed by Larsen and Yager in [Larsen93], is another approach. They use the *fuzzy relational thesauri* for representing the fuzzy implication between terms, in order to compute the *strength* among the query terms and document descriptors. GURU defines an indexing scheme based on lexical affinities of natural language software documentation and comments. It uses browsing technique to select software components. As we describe later, the classification schema adopted by our system is based on the degree of the semantic similarity between two multiwords phrases.

In this paper we describe PRASSY (Procedures Retrieval And Storage System using hYpertext), a hypertext system for the storage and retrieval of relevant procedures source code contained in programs developed by graduate students at the Computer Science Department at the University of Milan. PRASSY adopts the IR approach, based on the semantic analysis of natural language comments within the program code. Moreover, it uses the browsing mechanism of hypertext technology to search reusable software components. As asserted by Maarek et al., since there is rarely a component of software libraries perfectly matching a user's query, browsing is an adequate operation to single out elements from these libraries [Maarek91].

A prototype of PRASSY has been developed and runs on a HP9000/300 workstation with the HP/UX Operating System. This paper presents the system architecture and functionalities of PRASSY with emphasis on the browsing mechanisms. Some examples of the user interface are reported. Finally, the algorithms adopted for defining the semantic similarity between phrases expressed in natural language are described. These algorithms are fundamental for the construction of the hypertext and for the retrieval of procedures stored in it.

2 PRASSY basic elements.

Much work has been done on the hypertext concept. In the area of software engineering, some systems, such as Context[Delisle87] and Neptune[Schwartz86], are proposed to support the design and documentation process of large-scale software applications. Moreover, hyperCASE [Cybulski92] is an integrated environment of CASE tools where hypertext technology is used for linking and browsing fragments of any type of information.

PRASSY is not the complete answer to support the collaboration among software engineering teams. It refers to the limited question of how to classify C subroutines on the basis of the comments associated with them and how to make use of their source code for the software developers. By means of PRASSY, the subroutine source code is inserted in a hypertext structure and it can be located using the hypertext browsing. As for the system proposed by Prieto-Diaz and Freeman, PRASSY provides a way for measuring the semantic similarity between the user requirements and the candidate node to be selected. At the present time, programs archived by PRASSY are coded in C. Modifying PRASSY for managing programs coded in other programming languages requires further work.

PRASSY does not consider a program as an inseparable object but as a structured object, whose components are procedures of the program itself. The classification scheme adopted by PRASSY is based on the semantic similarity between the comments inserted into the program procedures. In fact, comments are the unique instrument integrated in the source code capable of describing, even if only partially, what a procedure performs.

PRASSY considers only the comments delimited by two particular markups, ".SPR" and ".END", which represent the start and the end of the procedure description respectively. We will refer to this piece of comment as *descriptor*. A descriptor can be constituted by more than one phrase (i.e., a sequence of words terminated by a full stop). We will refer to each phrase of the descriptor as *descriptor phrase*. Inserting markup in existing programs needs additional editing work on their codes, but this is necessary for isolating significant information among several comments. Indeed, almost all of the comments are assertions used by programmers for describing the data structures and are not related to the work performed by a procedure.

One of the main works to build a hypertext document is the link definition. According to the type of the link, there are several proposed techniques. One technique is the automatic linking by interpreting structural information of the document, e.g., typographic tag elements. Another approach to define link elements is the text semantic analysis. Bernstein mentions systems called *shallow apprentices* which are, as defined in[Bernstein90], systems which discover links through superficial textual analysis without attempting to analyze meaning. Principally, the approach adopted by PRASSY to define links could be considered similar to the shallow apprentice technique.

The position of a procedure in the PRASSY hypertext depends on the meaning of its descriptor respecting to the other ones. The value of the semantic similarity between descriptor phrases is established on the basis of the algorithm proposed by Aragon-Ramirez and Paice[Paice85] (we will abbreviate the author names as "ARP"). This value is a real number R in the range [0.0-1.0]. The semantic of two multiword-phrases S and T are considered perfectly similar when their R(S,T) value is equal to 1.0. We will describe the ARP's algorithm later.

Information treated by the hypertext is organized in the following way:

1. Each procedure stored in the hypertext consists of the respective descriptor and source code useful for reuse.

2. Procedures whose descriptor phrases produce the R value exactly equal to 1.0 are grouped into the same node. The first procedure inserted in a node is called *master procedure*, while the remaining procedures of the same node are referred to as *cluster procedures*.

3. Links to a node are represented by R values smaller than 1.0. That is, procedures of a node and those of its neighbours are not perfectly similar.

4. If a descriptor is constituted by several phrases, these descriptor phrases are treated separately. In this way, the same comment, and consequently the same procedure, can be collocated in more than one node. In other word, we do not consider the relationships that could exist between phrases of a same comment. We think that in a comment, every phrase usually expresses concepts completely independent from each other.

3 Phrases analysis.

Establishing the similarity between phrases in natural language is a difficult task. Salton presents several solutions in[Salton89]. As mentioned, the approach adopted by PRASSY derives from the ARP's algorithm. It establishes that the semantic proximity between two phrases depends on the degree of similarity between each word of the first phrase and the possible word of the second phrase associated with it. The algorithm does not consider the meaning of the phrases.

To establish this similarity degree, PRASSY constructs a thesaurus containing similar words on the basis of statistical calculation. The work emphasizes the fact that two words are considered semantically similar if they have a high number of identical initial letters, or if they are simultaneously present in different phrases. It also attributes importance to the number of occurrences of these words in the pattern documents. The calculation uses an approach similar to the vector-space model and the Jaccard coefficient[Salton89] for normalizing the similarity values in order that they can be inclusive between 0 and 1.

A similar technique is used in the GURU system. In this system indices are constructed on the basis of the lexical affinities between two words, determined by their common appearance in the same document. Instead, in the Prieto-Diaz'system the thesaurus is a vocabulary that groups synonyms under a single concept, while the degree of similarity between terms are expressed by the conceptual graph.

Once the thesaurus is constructed, it is possible to determine the similarity between phrases using ARP's algorithm which is represented by the following formula:

$$R(S,T) = 1/m \sum_{x=1}^{m} W_x.V(S_x,T_y).I(J)$$

where:

- m is the number of words of the longest phrase between S and T. Let's suppose that it is S;
- $V(S_x, T_y)$ is the semantic proximity value between S_x and T_y. The V value is calculated on the basis of values present in the thesaurus described above;
- y is the position of a T's word associated with the x-th word of S. Rules established by ARP indicate that each word of S may be mapped at most onto one distint word of T. The mapping is established by an appropriate function y=J(x). From the implementing point of view, for the x-th word of S, y represents the position of a T's word whose V value respecting S_x is the hightest among the V values of all words of T;
- R (S,T) is a real value in [0.0-1.0], normalized by the factor 1/m;
- W_x is the weight associated with S_x. We have assigned the value 1 to all words of S;
- I(J) establishes the order-matching between the associated words of S and T. In fact, the calculation of R is influenced by their positions in the corresponding phrases.

4 The PRASSY system.

A prototype of PRASSY has been developed utilizing the C language. The implementation of the user interface component is supported by OSF/Motif R1.1 widget set, based on X Window System Version 11 Release 4.

As pictured in Figure 1, the functional architecture of PRASSY is divided into two principal parts, corresponding to two separate phases of PRASSY utilization. The first phase is related to the construction of the thesaurus. In this phase the proximity values of pairs of words are registered in an apposite library. As described above, similar words are established by recognizing those with identical initial letters and by statistical calculation. For this job the manual pages of several UNIX commands are provided to the system as pattern documents. Before the hypertext is built, PRASSY allows the user to update the constructed thesaurus, adding or deleting words and modifying the V value of every pair of words. The second phase concerns the definition and use of the hypertext (Figure 2) . To resolve the disorientation problem for the user during the browsing[Brown89], the hypertext is designed with the following characteristics:

- it corresponds to a simple graph structure (nodes represent similar procedures and links represent the semantic proximity values with neighbour nodes)
- it presents information regarding the graph in the textual form
- it uses different open windows for representing different paths followed by the user who can switch from a path to another one at any moment
- it provides for every path a history list which allows the user to go back to some node.

Three main operations can be performed on the hypertext: node administration, browsing, and procedure code reuse. They are represented in detail by Figure 2 and described in the next sections.

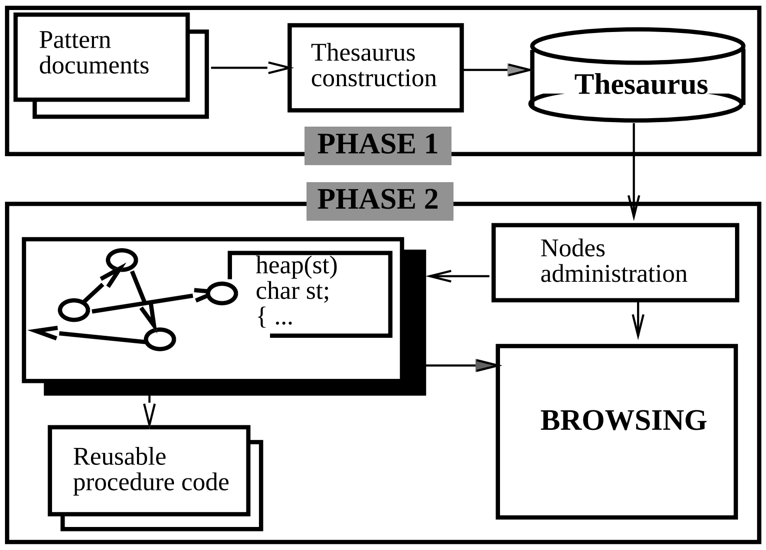


Figure 1: The architecture of PRASSY

4.1 Node administration.

The hypertext administration activities consist in inserting/ deleting some procedures in/ from the hypertext. Note that because of the hypertext security, PRASSY users are divided into two classes: programmers and hypertext administrators. The former can propose the procedure codes to be archived and look for procedures to reuse, but they can't modify the hypertext. On the other hand, the hypertext administrators analyze procedures from the proposal list built by authors of the programs and decide whether or not they should insert them in the hypertext.

The programmer's operations to propose new reusable procedures are represented in the Figure 2 and are outlined in the following steps:

1. Descriptors analysis (step 1): When the programmer thinks that his own program could contain useful procedures to be reused, he needs to examine the source code for selecting the most interesting ones. By parsing the C file indicated by the programmer, the system extracts from the source code the comments enclosed between the markups ".SPR" and ".END" and shows it to the programmer.

2. Host node search (step 2): For every phrase of these comments, the system searches for the most adequate hypertext node which can put the new procedure up. This work is carried out calculating the R value between such phrase and the master procedure descriptor of each hypertext node. There are two possible cases, according to the R value obtained. If there is a node which produces the R value equal to 1, it will put the new procedure up. The procedure would become a cluster procedure of the considered node. We remember that each node may contain more than one procedure. Otherwise, a new node must be created, where we can insert the new procedure. In this case the procedure will be considered as the master procedure of the new node.

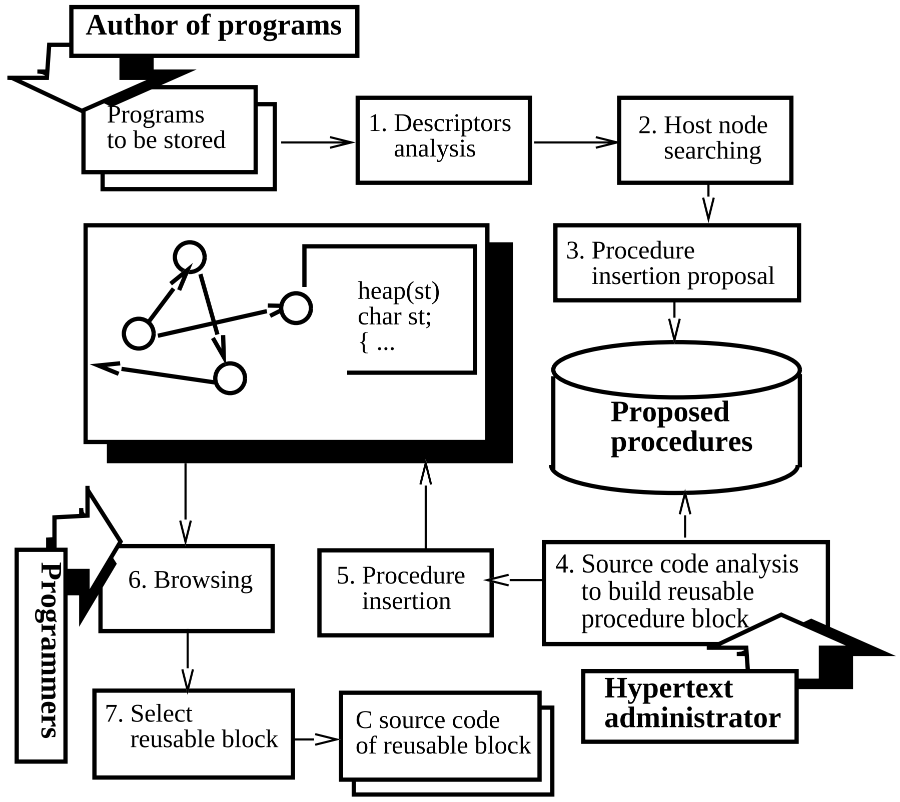


Figure 2: Steps to use the PRASSY hypertext

3. Procedure insertion proposal (step 3): Based on the results of the previous phase, the program's author can confirm the intention to insert the new procedure into the hypertext. In this case, information regarding the procedure are appended in an appropriate proposal list. No node is created during this phase because, as mentioned, only the hypertext administrator is able to modify the hypertext.

The two successive steps (4 and 5) concern the hypertext administrator who can examine each element of the proposal list and confirm whether or not it should be inserted in the hypertext. In the case of confirmation , what PRASSY has to do becomes a bit more complex. In fact, inserting a new procedure into the hypertext means registering its descriptors together with its source code. But often the procedure uses external objects that are not present in its own definition section, for example global variables, header files, called procedures, macros. This reliance on non-local objects does not facilitate an possible successive reuse of the procedure. As we describe later, PRASSY is able to group all objects referred by the procedure into a unique file, called *reusable block*, in order for it to be able to work in a autonomous way during the reuse phase.

4.2 Searching procedures.

The main objective of PRASSY is to select procedures that contain the most adequate solutions for some specific programmers' problems. The user has two ways of navigating in the hypertext and he can use them simultaneous (step 6 of Figure 2):

1. By means of a query which allows the user to express the topic of his research in a natural language. The system will direct him to a node which contains the procedures whose descriptors are semantically closed to his query,

2. By means of the browsing mechanism which allows the user to navigate among hypertext nodes until the required procedure is obtained.

Since browsing is an important function of the system, we will describe its user interface in detail later.

4.3 Procedure source code reuse.

Once the procedure of major interest is selected, the user can enter in a new environment for analyzing its source code in detail. (referring to the seventh step of Figure 2). When the procedure is inserted in the hypertext, all of the objects it refers to are grouped in a *reusable block*. The content of the reusable block is displayed to the user in an appropriate window. It is divided in data structures section and code section. This allows a careful procedure data structure analysis. Researches carried out by Meyer[Meyer87] confirm that updating the data structure represents 80% of the work needed to adapt a reusable component to the new programmer's requirements. If the code of the reusable block shown is considered useful by the user, it can be saved as a C source file and used for new programs.

4.4 Miscellaneous.

Besides the above functions, the system also provides set-up commands to establish the users' working environment. These functions allow the user to specify a set of default values, such as the number of browsing operations to be memorized in the history list or the minimal acceptable proximity value for selecting a procedure.

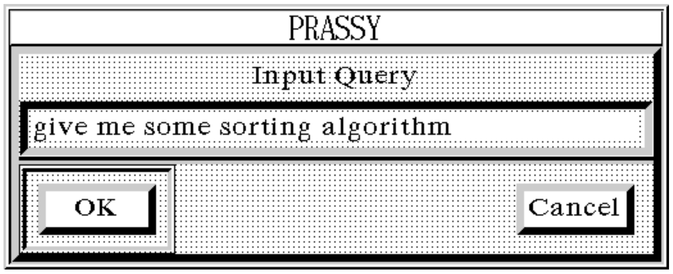
Moreover, at any moment, the user can obtain the history list which reports the sequence of nodes that he has selected up to that point. The number of nodes to be displayed is established by the user using the set-up functions. By selecting one of the procedures present in the history list, the user goes to the corresponding node and can continue to navigate in the hypertext. Note that every browsing session has its own history list. At last, the back step function allows the user to go back, step by step, through the path he has used.

5 PRASSY user interface.

The user interface of PRASSY is rather simple and uses the most elementary OSF/Motif widgets, such as pop up and pull down menus and dialog box. The main functions of the application are: *System* for setting up default values, *Archive* for managing hypertext nodes (insert and cancel nodes) and *Help* for showing on-line helps.

The remaining functions of PRASSY are obtained by means of a pop-up menu which contains five buttons. The first one, whose use mode is described in the next section, is the most important and represents the *query* function which allows the user to navigate in the hypertext. The remaining buttons correspond respectively to the history function, the reusable block source code analysis function, the back step function and the multipath function.

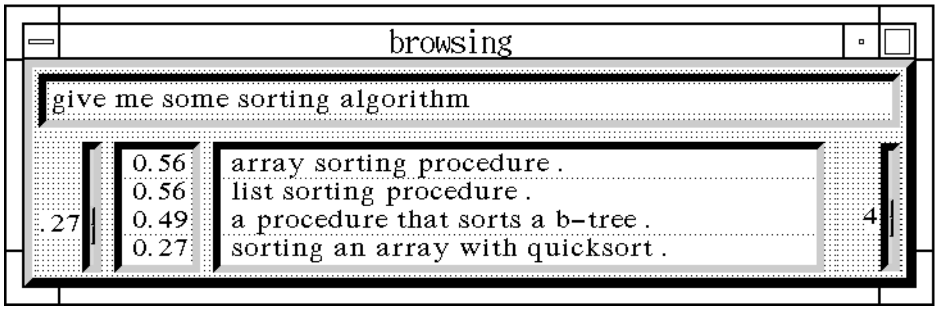
Selecting the *query* function, a dialog box will appear asking the user to introduce the query. An example is shown in Figure 3, where the users' question regards some sorting algorithms.



give me some sorting algorithm

Figure 3: The query dialog box

As a result, the system will present a list of nodes containing those procedure descriptors that have proximity values, with respect to the user query, over a given default threshold value. We will refer to these nodes as *adjacent nodes*. The answer to the previous user query on the sorting algorithms is reported in Figure 4, where the master procedures of four adjacent nodes are shown. The figure represents also a typical window for browsing



give me some sorting algorithm

0.56 array sorting procedure .
0.56 list sorting procedure .
0.49 a procedure that sorts a b−tree .
0.27 sorting an array with quicksort .

Figure 4: A typical window for browsing functions

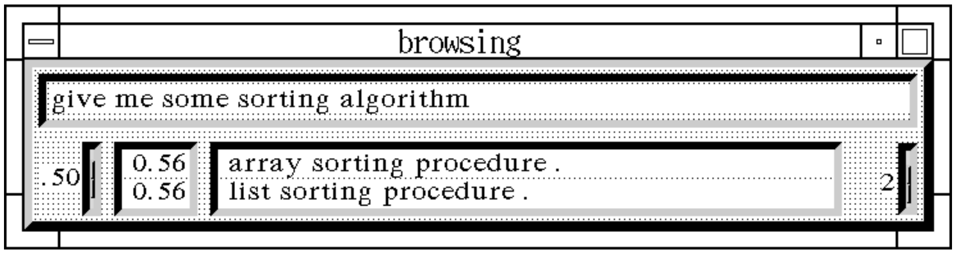
operations. The window contains the following important elements:

- a title window, containing the current node name or the query phrase;
- a main window containing the list of adjacent nodes;
- a vertical scroll bar (at the extreme left) which allows the user to regulate the threshold value (in Figure 4 it is 0.27). We will refer to it as *quality bar*;

- a vertical scroll bar (at the extreme right) which allows the user to fix the maximum number of master procedures to be displayed. We will refer to it as *quantity bar* (in Figure 4 this value is 4);
- an optional window indicating the proximity value of every element of the list respect to the current node.

The title window, which usually represents the current node, can sometimes contain the query phrase. This does not confuse the user because he can consider the query phrase as a virtual node from which begin browsing operations.

As mentioned above, a link can have a value in the range between 0.0 and 1.0, corresponding to the proximity value between two nodes. The quality scroll bar represents this range of values. By scrolling the bar, the user can change the minimal proximity value that adjacent nodes (listed in the window) must have with the current node. In this way, if the user wants to identify a very small set of procedures which are strictly related to his research, he can raise the threshold value by moving the bar up, for example until 0.5. Consequently, as reported in Figure 5, only two procedures satisfy the user requirements regarding sorting algorithms.



give me some sorting algorithm

```
0.56  array sorting procedure .
0.56  list sorting procedure .
```

Figure 5: The use of the quality scroll bar up

On the other hand, if he wants to explore the net, he can lower the threshold value, so that nodes far from the current one can also be listed (see Figure 6).

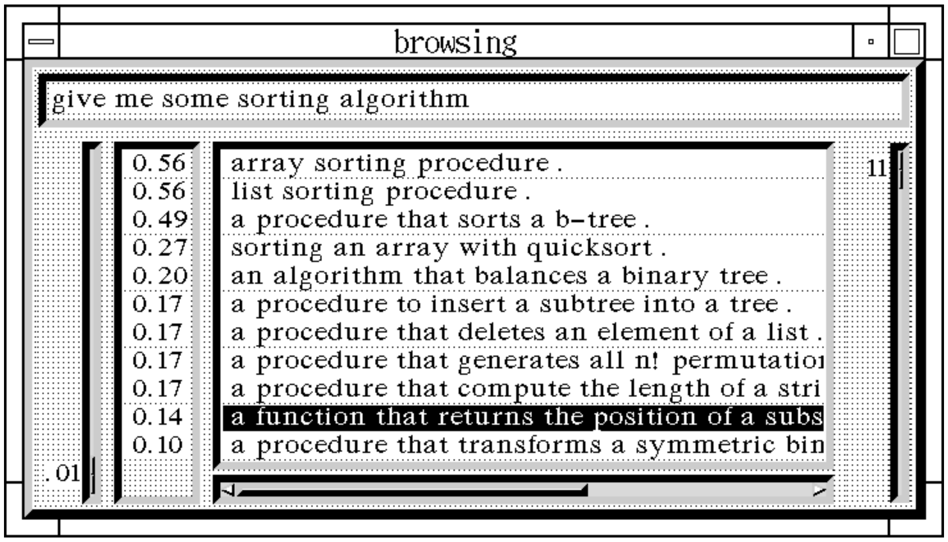
By clicking on one of the elements in the procedure list (e.g., the penultimate one of Figure 6), it is possible to move into a new node. The procedure selected becomes the current node, while the window displays a new list of procedures of its adjacent nodes (Figure 7).

In the above example, we have moved to a new research topic, going from a hypertext node representing sorting algorithms to a new one representing string manipulations.

The multiwindows characteristic of X Window system allows PRASSY to have more than one browsing window present simultaneously on the screen. By chosing this function, a new browsing window identical to the current one is created. The user can move from a window to another one, following different paths to explore the hypertext.

Moreover, modifying the application configuration file, the user is able to express his intention to iconize the current browsing window when a new one is created. All icons are lined up in a determined position of the screen. By selecting one of them the corresponding window will appear.

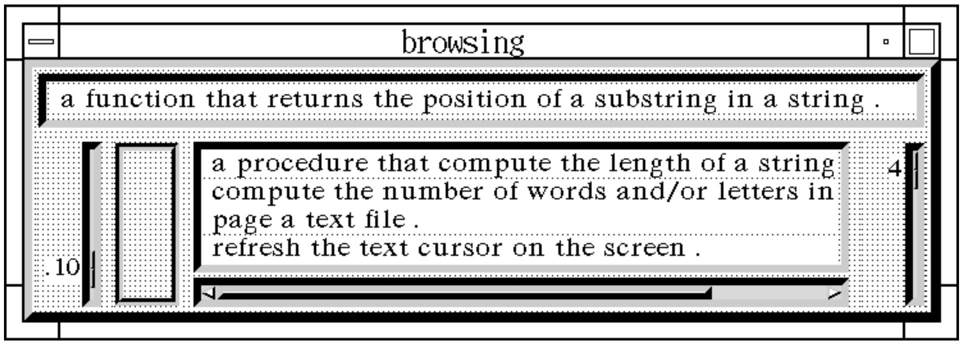
At last, once the user has selected the interested procedure descriptor from the browsing window, he can ask the system to display the relative reusable block source code.



give me some sorting algorithm

```
0.56  array sorting procedure .
0.56  list sorting procedure .
0.49  a procedure that sorts a b-tree .
0.27  sorting an array with quicksort .
0.20  an algorithm that balances a binary tree .
0.17  a procedure to insert a subtree into a tree .
0.17  a procedure that deletes an element of a list .
0.17  a procedure that generates all n! permutatioı
0.17  a procedure that compute the length of a stri
0.14  a function that returns the position of a subs
0.10  a procedure that transforms a symmetric bin
```

Figure 6: By clicking on one of the elements in the procedure list, it is possible to move into a new node



a function that returns the position of a substring in a string .

```
a procedure that compute the length of a string
compute the number of words and/or letters in
page a text file .
refresh the text cursor on the screen .
```

Figure 7: The new node which represents the new research topic

6 Conclusions.

During the testing phase of PRASSY we have analyzed a limited set of pattern C programs (about 50) coded by our students, obtaining about 200 meaningful procedures to insert into the hypertext. The main problem we had was the fact that not all programmers describe adequately, by means of comments, the functionalities of the procedures they build. Other factors that influence the correct answer of the system to the user query are the completeness of the thesaurus and the reliability of the algorithm adopted to calculate the proximity values between words of the thesaurus itself. While the algorithm adopted could be considered sufficiently reliable, we think that we need to increase the amount of thesaurus words.

Finally, with adequate updating, PRASSY could be used in an object oriented programming environment for classifying classes of objects, on the basis of their properties. Indeed, some of the main characteristics of the object oriented programming technique, such as data hiding, objects hierarchy definitions, and the polymorphism (methods homonymy) allow a more flexible way of reusing existing data and procedures (methods). In fact, data hiding means considering objects as *black box* to be put together for new projects. Classes are defined by expanding concepts from general to specific. New classes can be built by inheriting the characteristics of existing classes. These elements are under study for a new version of PRASSY.

Acknowledgement.

The author would like to thank the staff of the Text Processing Laboratory and the graduate students at the Computer Science Department at the University of Milan for their valuable support during the development of the PRASSY project.

References

[Krueger92] Krueger Charles W., Software reuse, ACM Computing Surveys, Vol 24, N 2, June 1992, pp. 131-183.

[Prieto-Diaz89] Prieto-Diaz Ruben, Classification of Reusable modules, in Software Reusability: Volume I-Concepts and Models, Biggerstaff, T.J., and Perlis, A. J., Eds. ACM Press, New York,1989, pp. 99-123, Chap. 4.

[Luckham84] Luckham D.C., and Von Henke, F.W., An overview of Anna, a specification language for Ada, in 1984 Conference on Ada Applications and Environments, St. Paul, Minn., Oct, IEEE Computer Society Press, Los Alamitos, Cal, pp. 116-127.

[Larsen93] Larsen Henrik .L. and Yager Ronald R., The Use of Fuzzy Relational Thesauri for Classificatory Problem Solving in Information Retrieval and Expert Systems, IEEE Transactions on Systems, Man, and Cybernetics, Vol 23, No 1, Jan/Feb 1993, pp. 31-39.

[Omohundro93] Stephen M.Omohundro, The Sather 1.0 Specification. Technical Report, International Computer Science Institute, Berkeley, Ca., Sept. 1993.

[Katz92] Katz S., Richter,C. A., and The, K., PARIS: A system for reusing partially interpreted schema, in Software Reusability: Volume I-Concepts and Models, Biggerstaff, T.J., and Perlis, A. J., Eds. ACM Press, New York,1989, pp. 257-274, Chap. 10.

[Adams93] Sam Adams, Ed Seidewitz, Brad Balfour, David Wade and Brad Cox, Software Reuse, (Panel Session), in Proc. OOPSLA Eighth Annual Conference, 26 Sept-1 Oct 1993, Washington, DC, pp. 137-143.

[Bassett87] Paul G. Bassett, Framed-Based Software Engineering, IEEE Software, Vol. 4, No. 4, July 1987, pp.9-16.

[Jones84] T.C. Jones, Reusability in Programming: A survey of the State of the Art, IEEE Trans. Software Engineering, Vol. 10, No. 5, Sept. 1984, pp.499-493.

[Redwine89] Redwine Jr. S. T., W. E. Riddle, Software Reuse Processes, Software Engineering Notes, ACM SIGSOFT, Vol. 14, No. 4, 1989, pp.133-135.

[Kaiser87] Gail E. Kaiser, David Garlan, Melding Software Systems from Reusable Building Blocks, IEEE Software, Vol. 4, No. 4, July 1987, pp.17-24.

[Prieto-Diaz91] Ruben Prieto-Diaz, Making Software Reuse Work An Implementation Model, ACM SIGSOFT, Software Engineering Notes, Vol. 16, N. 3, July 1991, pp.61-68.

[Lenz87] Manfred Lenz, Hans Albrecht Schmid, Peter F. Wolf, Software Reuse through Building Blocks, IEEE Software, Vol. 4, No. 4, July 1987, pp.34-42.

[Tracz90] Will Tracz, Where Does Reuse Start, ACM SIGSOFT, Software Engineering Notes, Vol. 15, N. 2, Apr. 1990, pp.42-46.

[Maarek91] Yoelle S. Maarek, Daniel M. Berry, Gail E. Kaiser, An Information Retrieval Approach for Automatically Constructing Software Libraries, IEEE Transactions on Software Engineering, Vol. 17, N. 8, Aug. 1991 pp. 800-813.

[Burton87] B. A. Burton, R. Wienk Aragon, S.A. Bailey, K.D. Koelher and L.A. Myes, The usable software library, IEEE Computer, Vol. 4, No. 4, 1987, pp. 129-137.

[Frakes87] W.B. Frakes and B.A. Nejmeh, Software reuse through information retrieval, Proc. 20th Ann HICSS, Kona, Jan, 1987, pp. 530-535.

[Delisle87] N. M. Delisle, M. D. Schwartz, Contexts- A Partitioning Concept for Hypertext, ACM Trans. on Office Inf. Sys., Vol. 5, N. 2, April 1987, pp.168-186.

[Schwartz86] M.D. Schwartz and N.M. Delisle, Neptune: A Hypertext System for CAD Applications, Proc. Intern . Conf. on Management of Data, ACM, Washington D.C., New York 1986, pp.132 -143.

[Bernstein90] Mark Bernstein, An Apprentice That Discovers Hypertext Links, Proc. Hypertext Conference 90: Concepts, Systems and Applications, Cambridge University Press, Paris, July 1990, pp. 213-223.

[Paice85] Paice C. D., Aragon-Ramirez V., The calculation of similarities between multi-word strings using a thesaurus, Proc. RIAO, Grenoble 1985, pp.293-319.

[Salton89] G. Salton, Automatic text processing, Addison-Wesley, 1989.

[Brown89] P.J. Brown, Do we need maps to navigate round hypertext documents?, Electronic Publishing, Vol. 2, N. 2, July 1989, pp.91-100.

[Meyer87] Bertrand Meyer, Reusability: The Case for Object-Oriented Design, IEEE Software, Vol. 4, No. 2, Mar. 1987, pp. 50-65.

[Cybulski92] Cybulski, Jacob L., Reed, Karl. A hypertext based software engineering environment IEEE Software v9, n2, March, 1992, pp. 62-68.