Automatic Goal Extraction From User Actions To Accelerate The Browsing Of Software Libraries.

C. Drummond, D. Ionescu, and R. Holte

Departments of Electrical Engineering and Computer Science
University Of Ottawa, Ottawa,Ontario, Canada, K1N 6N5

This paper describes an active browser which accelerates the searching of software libraries for reusable code. The system supports normal, static, browsing actions which permit a user to navigate a library by traversing predefined relationships between components. As an active browser, the system monitors and uses these actions to infer a template representing the user's goals. With this template the system scans the library independently of the user looking for potentially interesting components. One advantage of independent search stems from the fact that in any structured library similar items may be well seperated within this structure. We demonstrate this fact in an object-oriented library and how the system locates items which are remote, in the inheritance hierarchy, from the area being browsed.

2 Introduction

This research is centered on software reuse, widely accepted to be a significant way of improving software productivity. To many the ultimate goal is one of producing software components analogous to integrated circuits [1,2]. Certainly research into the design of reusable software has led to considerable activity in creating large software libraries. A prime example of which is object-oriented programming.

Large libraries, alone, are not a complete answer. Biggerstaff and Richter[3] highlight four critical issues in reuse: finding, understanding, modifying and composing components. We primarily address the first of these, considering it the most important.

Generating useful indices for items, either manually or automatically, is one way to accelerate search. In Prieto-Diaz [4] a component description based on facets, manually determined by rigorous domain analysis, is used to form tuples as indices. S. Meggendorfer and P. Manhart [5] adopt a more formal approach assigning a specification to each component. In S.D.Fraser et al. [6] keywords are automatically extracted from text associated with software. Y.S. Maarek et al. [7] automatically extract indices, through a more complex analysis of text, in the form of lexical affinities.

Even a good indexing scheme is not effective if the user's is unfamiliar with the indexing language. In the faceted approach this is partially answered by the addition of thesauri and special conceptual graphs. Yet even knowing the language, it may be difficult to formulate an accurate description of the requirements. Any match on the query is therefore unlikely to be exactly what is wanted. To answer this, in Maarek's system lexical affinities are used to cluster components and this allows limited browsing. One approach that aims to address both these problems is retrieval by reformulation. In G. Fischer et al.[8] a query is matched using spreading activation through a graph of keywords and library items where the arcs represent mutual association. The items with the highest activation are returned plus the keywords that were activated during the process. The user can modify the query by the selecting from these related keywords and continue search.

In reuse the solution depends on what is in the library and the available items may only offer partial solutions. The amount of search required is much greater than the previous systems support. Browsing as a means of search is an attractive alternative. To be used as a search methodology the main requirement is that the library can form a graph with the items as nodes and relationships meaningful in terms of search.

The system described here creates a form of automatic query reformulation within a browsing environment. Each time the user selects a new component and inspects its properties a dynamic template is updated. This template then retrieves components which best match the current search goal.

Section II of this paper describes the general architecture of the enhanced browser. Section III describes how templates are inferred from browsing actions. Section IV shows an example of how the system supports the user in search and section V presents the conclusions.

3 System Architecture

The system (see figure 1) consists of four main components, an object oriented library, a normal browser,a template matcher and a template inference mechanism. An object oriented library contains classes which consist of methods, instance variables and associated documentation. It also contains intrinsically the relationships among the individual items. Some are associated with control and data flow. Others are defined by the inheritance hierarchy or by the polymorphism of method names.

This information is made accessible to the user through a browser. Methods, instance variables, classes etc. are nodes in a graph. Application of operators results in the display of all nodes joined to the selected node by a particular type of relationship. The nodes are displayed in a list in one of the browser windows. Other operators may be applied to items in the new list and so on producing multiple lists.

The template models a class and consists of atomic terms, with associated weights, principally class, method and instance variable names. When matched against individual classes a graduated matching scheme compares

names broken down into constituent words. (addIfAbsent becomes add if absent). Matching on method names also includes methods implemented some distance down a calling path. If the method addIfAbsent was not implemented in the class but was used by a method in the class a lower matching score would be used. The template further includes methods, functions and global variables that might be used by methods in the class. Each class is assigned a value which is the normalised weighted sum of the degree of match with the terms.

Within this design the browsing actions are automatically passed from the normal browser to the template inference mechanism. The inference mechanism will pass a template to the matching mechanism for comparison with the whole library or one particular class. The first of these operations is independent search and results in an ordered list of classes with an associated value for its degree of match. The second allows the inference mechanism to compare the template with the class the user is studying. The results of this comparison is used to adjust the template to maintain its consistency with the current goal. The inference mechanism consists of converter which puts the browser actions into predicate form. Plus there are several rule bases, a working memory and inference engine. Finally there is a second converter to translate the template predicates into a usable template which can be matched against the library items. This is described in detail in the following section.

4 Inferring Search Templates

Browser actions received by the template inference mechanism are added to the history list which the system uses to look up previous actions. It is useful as a record of user search and can be analysed for interesting patterns. It is written to a file on exit from the browser. The action is also converted to a predicate of the form usable by the inference engine:-

(Node type, node, operator, prior node, prior operator).

The node type and node define where the user is in the program graph. The prior node and operator are the name of the node and operator that produced the list from which this item was selected. These are used to allow chaining of action predicates to identify particular search patterns.

An example is the sequence starting with the selection of the menu option superClasses. Applied to the Class named OrdCltn (Ordered Collection) this lists its ancestor classes. Applying the menu option subclasses to the item in this list Cltn (Collection) would display a list of its children, the siblings of OrdCltn. The second predicate produced is:-

(Class Cltn Subclasses OrdCltn Superclasses)

OrdCltn is the prior node and Superclasses the prior operator and will be identical to the node and operator of the previous browsing action. It is not necessary that these two operators be consecutive. The user may make other menu selections after the first and then return to the ancestor list for the next selection. The prior node and operator allow the system to determine this and thus detect the pattern of the two operators. The chaining of predicates can be extended indefinitely by this means. A longer sequence will be shown in the example.

Meta-rules use the node type to select the subgroup of rules that are appropriate for the action. The inference process is activated and results in predicates being added to working memory. These are of two forms: those which define the state of search and the template predicates. The latter are combined into a code template which can be sent to the browser for matching.

The rules are based on two basic methods of inferring a template. The first approach is the generalisation of classes visited as part of search. Here the properties of two or more classes are generalised to form a single template. This occurs for instance when the user explores classes that are siblings. The system infers that the user is interested in the common properties of the classes and adds the methods of the parent class to the template. This case will be detailed in the example. The second is an incremental approach where the user shows interest in a particular property. If the property is a method the name of the method is added to the template. As the user explores this and other classes more items are added to the template. An example of the incremental approach is the inference rule:-

IF ((Class ?name ?op ?pnode ?pop)(NOT interestedIn Class ?name)) Then ((Template className ?name 0.5)) 0.10

This rule means that if an operator is applied to a class, that the user has not showed interest in previously, then the template should include the class name. If this was matched against the library, using OrdCltn all items that contain Ord or Cltn as part of their names would be returned.

A confidence factor is associated with each of the rules and predicates. These factors are combined together to form new confidence factors for predicates produced by the inference process. For template predicates this number is multiplied by a scale factor, the number seen in the consequent of the above rule, to form the matching weight. The confidence and therefore the matching weight is increased for each operator applied to a node. For instance the confidence in the interest in a polymorphic method name is increased if it is studied in multiple classes.

As discussed in the previous section to maintain the templates consistency with the current search goal it is compared to the new class. The result of the match is returned to the reasoning mechanism where it is used to update the template using rules like the one shown below:-

IF ((TempMatch className:wt: ?type ?name)) THEN ((interestedIn ?type ?name) (Template+ className:wt: ?name 0.2)) 0.10

Before the comparison occurs the confidence of the template predicates is reduced. This produces an overall increase in the factors that match strongly with the new class and an overall decrease in those that do not match or only match weakly.

5 Computer Assisted Browsing

This section gives a simple example of the process in operation using a small commercially available library produced by the Stepstone Corporation. In figure 2 the history report shows the user's browser actions. It starts with the selection of a list of all methods locally defined (not

including those inherited) in the class named OrdCltn. Two methods "-add:" and "-find:" are studied in greater detail. The operator superclasses is applied to OrdCltn which returns the ancestor list containing Cltn. The operator subclasses is then applied to Cltn producing a list of its children. The class Set is selected, its methods viewed and again the method "-find:" is inspected. Figure 4 shows these actions converted to predicate form and the inferred template.

The template includes the names of the three classes visited and the methods inspected. The method "-find:" has a higher confidence factor than "-add:" as it was selected in both classes and examined more closely in the first. The template also contains commands to exclude the two classes which the user has already explored in detail. The second from last item shown in Figure 4 is added as a result of the first two rules shown in Figure 3. This implements the sibling search notion described in the previous section. The first rule shown produces the intermediate predicate "siblingsOf" on binding with the action whose operator is sub (subclasses) and whose prior operator is sup (superClasses). The confidence of the predicate is dependent on the confidence of the interest in the first class OrdCltn. The first clause of rule 2 determines if the operation is on a class node. This, combined with the binding of the "siblingsOf" clause implies the user is selecting from a list of siblings. The fourth clause makes sure the class selected is not the same as the original class. The resulting template item directs the template matcher to compare classes with all methods implemented in, or inherited by, the parent class Cltn.

The result of this template being matched is shown in figure 5. Cltn is the highest, as it is the parent class and also implements both "-find:" and "-add:". The third sibling, as yet unexplored by the user, is not the next best match. There are two higher scoring classes, SortCltn (Sorted Collection) and IdArray (Array of class ids) situated in unrelated parts of the library. The only common ancestor is Object the root of the inheritance tree. The high score is due to the extensive polymorphism deliberately designed into the library. Both classes are intended fulfill similar roles to the children of Cltn but because of their implementation differences are remote in the hierarchy.

6 Conclusions

This paper has discussed how a template may be inferred directly from normal user browsing actions and thereby enhance the browsing process. It has shown how a rule base can be constructed which contains knowledge to interpret the user's actions in terms of particular search patterns.

This paper has shown that the system can help the user locate relevant code that is not necessarily close in the inheritance hierarchy. This is a not uncommon occurence in object oriented languages as the hierarchy is dependent on exact implementation.

An important lesson learnt during this process was the problem with ambiguous actions. To facilitate the automatic inference of user goals the user interface must be altered to improve communication. Ambiguous actions should be replaced by functionally equivalent unambiguous ones.

We believe we have demonstrated that inference from user actions is a viable approach in driving systems that aid the user. Although this research has focused on object oriented libraries it is felt that this methodology could be readily extended to any browsing task i.e. searching for documents in a document base. Further the notion of direct inference from user actions should have a much wider applicability.

7 Bibliography

1. J. Rice, H. Schwetman Interface Issues In Software Parts Technology Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis Pub. ACM Press
2. D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, J. Garza Construction Of File Management Systems From Software Components. Technical Report TR-88-36 Dept. Of Computer Sciences University Of Texas
3. T.J. Biggerstaff, C. Richter. Reusability Framework, Assessment And Directions Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis Pub. ACM Press
4. R. Prieto-Diaz. Implementing Faceted Classification For Software Reuse. Communications of the ACM Vol 34 1991
5. S. Meggendorfer, P. Manhart. A Knowledge And Deduction Based Software Retreival Tool. Proceedings Of The 6th National Knowledge Base Software Engineerings Conference 1991
6. S.D. Fraser, J.M. Duran, R. Aubin. Software Indexing For Reuse. Proceedings of 1989 I.E.E.E. International Conference On Systems, Man and Cybernetics
7. Y.S. Maarek, D.M. Berry, G.E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. I.E.E.E. Transactions On Software Engineering Vol. 17 No. 8 Aug. 1991
8. G. Fischer, S. Henninger, D. Redmiles. Cognitive Tools For Locating And Comprehending Software Objects For Reuse. Proceedings of CHI-91 Human Factors In Computing Systems 1991

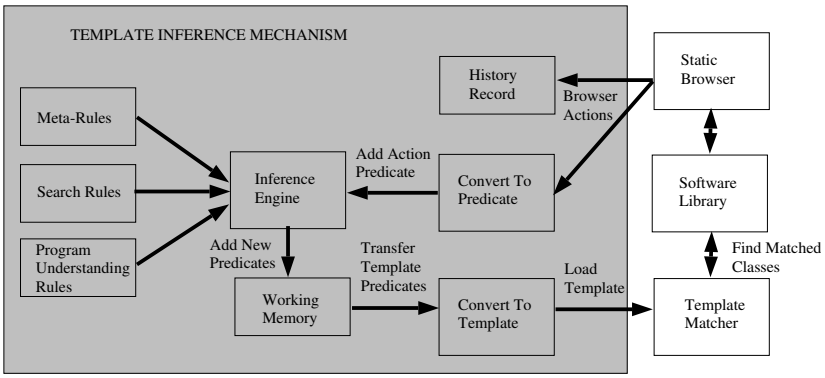

Figure 1 Browser Architecture

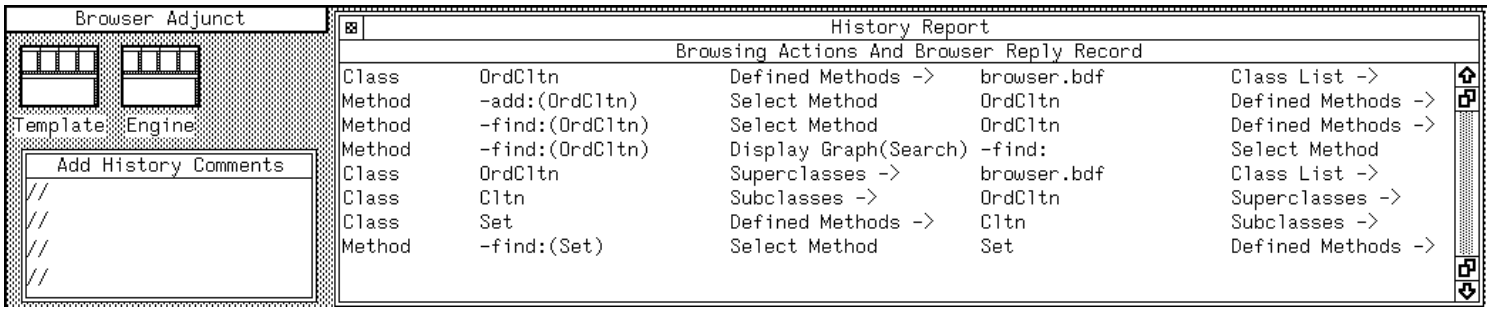

Figure 2 History Report

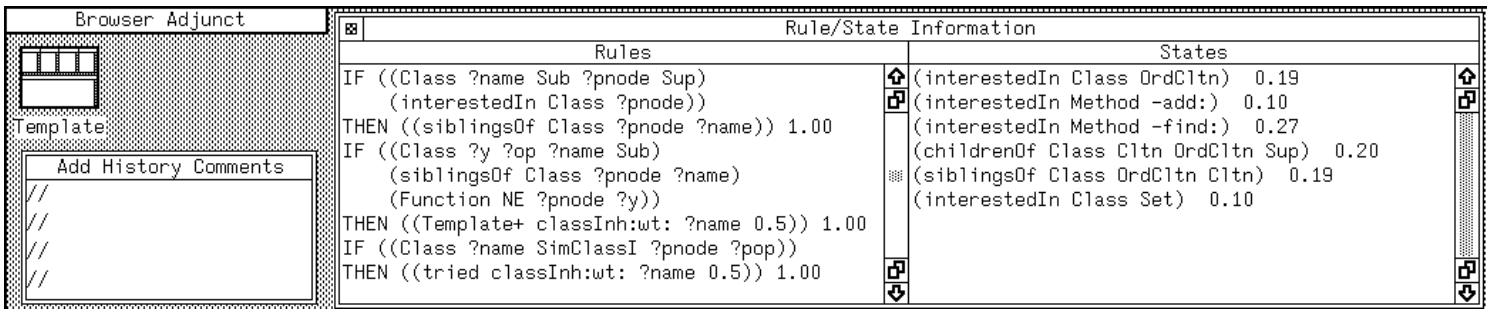

Figure 3 Rules And Intermediate Predicates

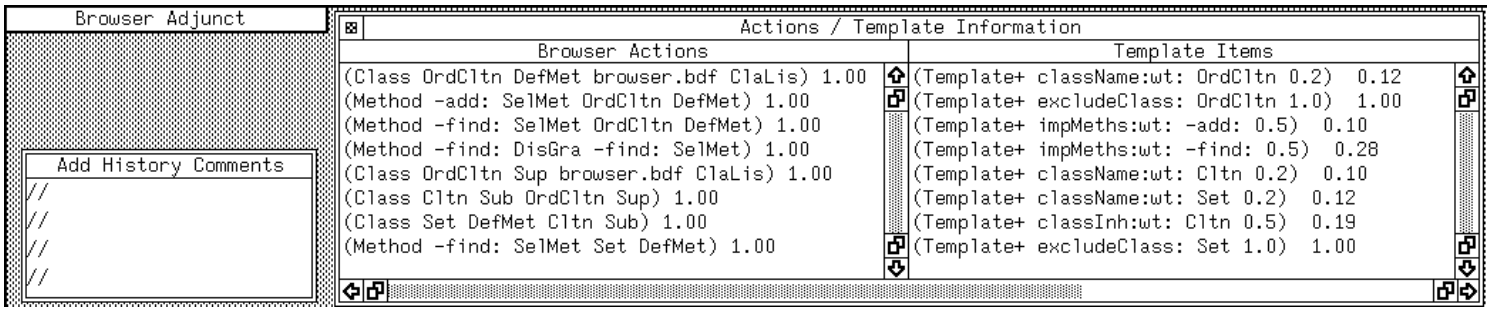

Figure 4 Browser Actions And Template Predicates

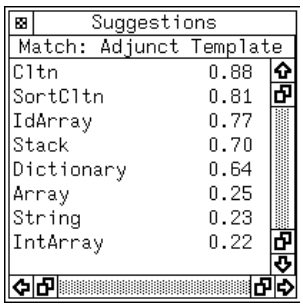

Figure 5 Suggestions