Improving Active Browsing with the Negative Inference and Selective Search Methods

by

John Ng Yuen Yan

A Thesis Submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of

Computer Science

under the auspices of

The Ottawa-Carleton Institute for Computer Science

Computer Science Department

Faculty of Science

University of Ottawa

Ottawa, Ontario, Canada K1N 6N5

ISBN 0-612-19998-3

Canada

Name NG YUEN YAN, John

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

EDUCATION
- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

SOCIAL SCIENCES
- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Methods 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

EARTH SCIENCES
- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

Applied Sciences
- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

PSYCHOLOGY
- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgments

I am greatly indebted to my supervisor, Dr. Robert C. Holte, for his guidance and support in the course of this research. I also want to thank him for his infinite patience and for his availability. My thanks go to Christopher Drummond who always made the time to reveal the "secrets" of Active Browsing to me in the early phase of this research, and to Dr. Denys Duchier for using his Spreading Activation code.

I thank members of the Machine Learning and Software Reuse group headed by Dr. Stan Matwin and Dr. Rob Holte and particularly Dr. Matwin who acted as my thesis advisor in the absence of Dr. Holte. This research wouldn't have been possible without the NSERC strategic grant obtained by Dr. Stan Matwin and Dr. Robert Holte.

Last but not least, I would to thank my family and especially my wife for their constant moral support and encouragement.

Abstract

Active Browsing is a technique whereby a learning apprentice assists a designer in locating software artifacts in reusable software libraries by inferring the user's search goal from the user's normal browsing actions. The aim of this research is to improve the response time and success rate of Active Browsing. Two methods are proposed for this. The Negative Inference method improves the success rate of active browsing by producing a more accurate representation of the user's goal. The Selective Search method improves the response time of the learning apprentice by limiting the system's evaluation of the library to a fraction of the library.

The Negative Inference method adds finer-grained features to the system's internal representation of the user's goal and rules for negative inference (i.e. inferring features that the user is not interested in). The Selective Search method defines a technique for partitioning the library and a strategy, called a migration policy, which determine which items to evaluate.

An implementation of both methods, based around a browser used to explore object oriented code, is described. This implementation is used to validate experimentally both methods. With Negative Inference the active browser's success rate is twice that of the normal active browser, and it ranks the search goal much more accurately at all stages of the search. With selective search, the active browser achieves similar success rate while only evaluating a quarter of the library.

Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

[Drummond, 92] describes Active Browsing as a technique in which a learning agent assists a designer in locating software artifacts in reusable software libraries. In an active browser the browsing system plays a more active role. The active browser suggests to the designer items it estimates to be close to the target of the search, the designer's "search target".

The aim of this research is to improve the response time and success rate of Active Browsing. The response time is improved by the Selective Search method in which the active browser is constrained to evaluate only a fraction of the library in order to suggest items relevant to the search target. The advantage of Selective search is to reduce the time taken by the system to update its suggestions when new user actions are observed. The active browser's response time naturally decreases with increasing size of the library as the system evaluates more items. The improved response time would make active browsing more feasible in very large libraries. Selective search achieves this with little impact on the success rate of the inference system.

The success rate, defined as the number of times the active browser infers the search target before the user finds it, is improved by the "Negative Inference" method. The main advantage of our Negative Inference method is that our system generates a more accurate representation of the search target than in [Drummond, 92]. This added benefit is achieved at no additional cost to the user since there is no change in the active browser interface. The user can interact with our system and accept the system's advice or not. With this non-intrusive property any slight improvement is valuable since no inherent cost is involved.

The principles of Negative Inference and Selective Search are implemented and tested in the context of a Software Reuse system. More specifically the Smalltalk Object library serves as the reusable software library.

1.1 Software Reuse

Software reuse can be defined as the process of constructing new software systems by using existing software artifacts. It is widely accepted that software reuse is a most promising technology for improving software quality and productivity [Biggerstaff and Richter, 87]. Software reuse, if done well, can simultaneously offer improved time to market, increased software quality, and reduced costs of development and maintenance. Software reuse is not limited to code only. [Krueger, 92] surveyed a diversity of techniques related to reusing artifacts, including design structures, module level implementation structures, specifications, documents and so on.

The promise of software reuse has led to the creation of large software libraries. Object-oriented languages and OO libraries, for example, have been an important step on the way to reusability. In OO programming, systems are built using a bottom-up development of new objects which utilize an existing class library.

The reuse activity is usually seen as a three step process: searching, understanding and adapting [Biggerstaff and Richter, 87]. The availability of reusable libraries does not guarantee successful reuse. Component retrieval is at the core of object-oriented programming [Freeman, 83; Frakes and Nejmeh, 87]. The task of finding the right classes is aided by the help of software code browsers.

1.2 The Problem of Locating Software Components

Locating software items is difficult when searching in large, complex and continuously growing libraries. Several factors can influence how successful a user might be at locating a potentially interesting component. One obvious factor is the familiarity and size of the library. The success of reuse systems as a development tool is thwarted by an inherent conflict: to be useful, reuse systems must provide many software components, but when many components are available, finding and choosing an appropriate one becomes a difficult problem. The large number and diversity of objects increases the chance that an object related to a problem task exists. However, finding such an object among many is

difficult. Another factor that applies to both experts and novices is the degree to which they can specify the characteristics of the desired component.

Browsing is a natural and effective process for locating items in libraries when the user cannot clearly define the characteristics of the desired component. The ability of users to recognize what is required over the ability to describe it makes browsing a suitable form of dialogue for locating components in reuse systems. Also the ability to evaluate the displayed items at a glance [Bates, 86] allows the user to search through large amount of material rapidly.

1.3 Active Browsing

Active Browsing aims to improve the normal browsing process. An active browser is a *learning apprentice* defined by [Mitchell et al., 85] as a "class of interactive knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem-solving steps contributed by their users through their normal use of the system" (p. 573). In contrast to most learning apprentices where the system requires user's feedback to correct its inference, active browsing is entirely unobtrusive: it infers the user's search target from the user's normal browsing actions, without requiring any feedback from the user.

The active browser consists of two main parts, an inference system and a template matcher. The "analogue" is defined as the active browsing system's internal representation of the user search target. It consists of a weighted set of "features" of a library item such as class item names and method item names (for an object-oriented software library). The inference system infers the analogue from the user's browsing actions. The "template" is used to measure the relevance of an item to the analogue. The template matcher computes the degree of match between a given template and a class in the library. At each update of the analogue, the template matcher evaluates all classes in the library against the template generated. The highest scoring items are then displayed to the user in a special window called the "suggestion box".

Active browsing maintains the advantages of normal browsing by allowing the user to search in the usual way. It supplements the normal browser by assisting the user in

his/her search. In the experiments in [Drummond, 92] 40% of the time active browser inferred the goal before it was found by the user.

1.4 Improving active browsing

We identify two ways of improving active browsing. First, we improve the inference system by adding a new type of inference rules and by adding finer granularity to the analogue. Second, we improve the speed of evaluating all library items on the template. This is especially critical in large libraries. We introduce two novel methods to achieve these objectives, the method of negative inference, and the selective search method.

1.4.1 Method of negative inference

To improve the active browser's inference we introduce the method of "Negative Inference". This method consists of two improvements to the active browsing system: the use of finer grained features in the analogue, and the addition of "negative" inference rules, which draw conclusions of the form "the user is not interested in X".

The granularity of the analogue is the level of detail of the target that the features describe. We increase the granularity of the analogue by adding more detailed features. This is achieved by extracting sub-words of method names, since method names consist very often of several words concatenated together. These words are called subterms. For example, in method "at:put:" the subterms are "at:" and "put:".

We define positive features to be ones that the user is interested in and negative features to be ones that the user is not interested in. We introduce negative inference rules to infer negative features.

Our method combines inferring both negative features and subterms, and consequent refining of the analogue. Positive features of the analogue are pruned using the analogue's negative features. The resulting analogue is more detailed and accurate than in [Drummond, 92] which infers only positive features. Furthermore, inference of the "extra" information is obtained from the same set of browsing actions available to the user, and

without introducing any other requirements on the user, thereby realizing an improved accuracy at no added cost.

1.4.2 Selective search method

During the browsing session, after each browsing action the active browsing system updates both the analogue and the template. A complete update involves computing the scores of all items in the library on the new template terms; this process is called the "matching process". In the "selective search" method only a fraction of the library is selected for the matching process.

The purpose of updating every library item's score at every iteration is to find the ten best items that match the user's search target, as represented by the analogue/template. The selective search method achieves this by evaluating only a subset of all items in the library. Each item score consists of an "exact score" and a "delayed" component: the "exact" score is the score on template terms on which the item was evaluated, and the "delayed" component is an upper bound for the item's "potential score" on all unevaluated template terms. At each iteration, items are selected for evaluation based on both their "exact score" and their "delayed score". The updated items are ranked according to both exact and delayed score, and presented to the user in the suggestion box. Selective search speeds up the matching process while maintaining reasonable inference performance.

1.5 Summary of Results

For this thesis, a new active browser was built that incorporated many features of the original active browser [Drummond, 92] which we refer to as the *Base active browser.* The original intent of the work was to design a "generic" active browsing system, in which the library can be described as a graph and searched using the Guidar [Duchier, 93b] search language. The new active browser is written in Common Lisp and uses the Smalltalk object oriented library as the reusable software repository.

Empirical results showed that *Base active browser*, a reimplementation of Active Browsing, succeeded 23.4% of the time whereas in [Drummond, 92] the active browser

had a success rate of 40%. The difference in inference performance can be attributed to the different application library and the differences in the active browser implementation.

The methods developed in this thesis were implemented in a *negative active browser* and a *selective active browser*. The *negative active browser* added negative inference to the *Base active browser* while the *selective active browser* added selective search to the *negative active browser*. Our approach to improving active browsing was then validated by measuring experimentally the performance of each active browsing system. However, the objective of the experiments with each system is different. With the *negative active browser*, the objective is to measure the performance gained by negative inference, i.e. how much faster is the new active browser at finding the search target than without negative inference. On the other hand, with selective search we measure the loss in performance when the degree of selectivity (i.e. the percentage of the library not included in the search) is increased.

We follow the same empirical method as in [Drummond, 92]. The preferred method of evaluating interactive systems such as an active browser is to test the system's inference in a browsing session involving a human searcher. The number of times it finds the user search target before the human user is a good metric of the inference performance of the active browser. A practical alternative is to use an automated agent instead. This ensures repeatability of the experiments and consistency of the agent's search. However, this is only an approximation of the performance of active browsing in real world usage since real world usage can be difficult to simulate for reasons such as the variability in human users.

Results obtained show dramatic improvements in performance. With negative inference, on average, the active browser succeeded 52.6% of the time, which represents a more than double increase from 23.4% for the Base active browser. Results obtained for selective search were similarly positive, and clearly show that selective search is a definite improvement over active browsing. When evaluating only 28% of the library the selective active browser inference performance equals that of negative active browser. When a mere 10% of the library is evaluated the search performance was similar to the Base active browser.

1.6 Contributions

1. A method of "negative inference" has been devised to improve the inference effectiveness of Base active browsing while maintaining the non-intrusiveness property. Two improvements were made to the active browsing paradigm by:
    - the use of finer-grained features in the analogue.
    - the addition of negative inference rules.
2. A novel method, called "selective search" has been developed, whereby only a fraction of the library is evaluated in order to update the suggestion box. This method includes:
    - a method for partitioning the library into partially evaluated sets of items.
    - a strategy, called a migration policy, which answers for each partially evaluated set the following two questions: "How many items from this set should be evaluated?" and "Which items in this set should be evaluated?".
3. An active browsing system has been developed in LISP for the Objectworks™ Smalltalk object oriented code library. Both selective search and negative inference methods were implemented in this browser. An architecture is developed to decouple the library from the active browser and includes a system for accessing nodes as an implicit graph. The architecture includes:
    - a rule based inference system using OPS5 [Brownston et al., 85],
    - a browser user interface using the Garnet library [Myers, 90],
    - the use of Guidar [Duchier, 93a] a general graph search language.
4. Empirical results were obtained to study and demonstrate the effectiveness of the above methods.

1.7 Organization

In Chapter 2, the general method of active browsing is presented. The discussion centers around a re-implementation of the technique originally developed in [Drummond, 92]. Chapter 3 presents the empirical method of evaluating the performance of the active browsing system. It also presents a discussion on the issue of comparing active browsers and presents various metrics to help measure and compare the inference performance of

active browsers. Chapter 4 presents the method of negative inference and its application to the active browsing method. Empirical results are presented and the method of negative inference is compared to Base active browsing. Chapter 5 presents the general method of selective search and the particular algorithm implemented in the active browser. Chapter 6 presents the results of the empirical validation of the selective search method. In Chapter 7, a broad overview of related work is presented. Finally Chapter 8 concludes and discusses limitations of the present system and ways to overcome them.

Chapter 2

2. Active Browser

This thesis is based on the active browsing method. The work in this thesis involved a reimplementation of the original active browsing system, with some variations. Section 2.1 presents active browsing as introduced by [Drummond, 92]. Section 2.2 presents the graph view of the library upon which search is accomplished. The inference system, which is at the heart of the active browser, is presented in Section 2.3. Section 2.4 presents the template and the process of matching. Finally we present an example of a successful active browsing search in Section 2.5.

2.1 Active Browsing Model

In the normal browsing process, the user interacts with the library through a browser. The left-hand side of Figure 2-1 illustrates the process of browsing. The user's goal is to find the library item that best matches his search target. Browser actions return

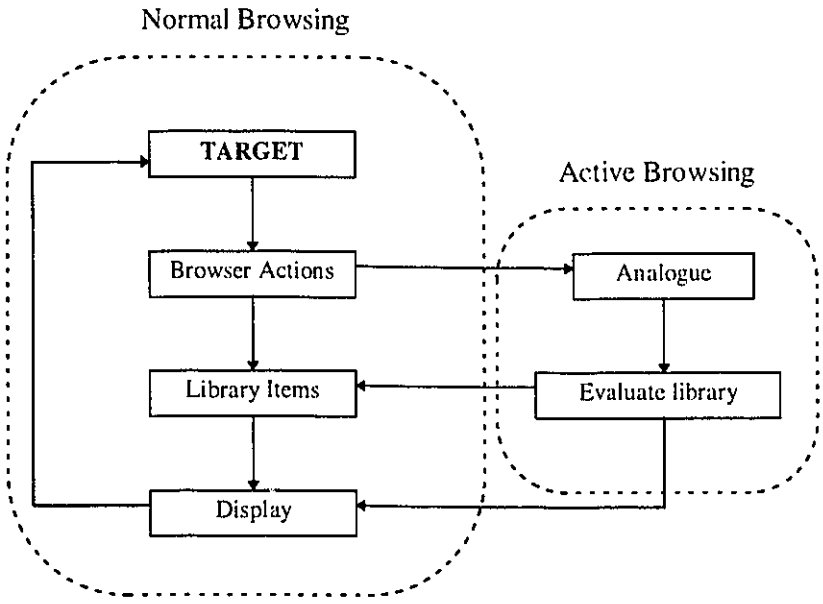

Figure 2-1 Active browsing

library items that are displayed in the browser, which are then compared by the user to his requirements, the search target. This process is repeated until the user is satisfied that the displayed item satisfies his requirements.

Active browsing adds a secondary loop to the normal browsing process. The active browser monitors the user's browsing actions and suggests interesting classes to the user. From the sequence of user actions, the active browser infers an "analogue" representing what it believes to be the user's search target. The analogue is converted into a "template", which is used to measure the relevance of a library item to the user. The template is matched against each item in the library, and the items sorted according to their relevance and displayed to the user in a special window (the "suggestion box").

2.2 Graph view of a library

In the active browsing paradigm, the library is viewed as a graph, where nodes represent library items, and graph links represent the relationships between library items.

2.2.1 Object Oriented library

The searchable library used in this thesis is a commercially available Object Oriented Software library, more specifically the ObjectWorks™ Smalltalk library. An object oriented library consists mainly of classes, each of which is a collection of private data and public operations. In general, a class has associated with it:

- a set of *variables* that contain the data for the object.
- A set of *messages* to which the object responds. These messages are the operations that can be performed on the object.
- A *method* which is a body of code implementing each message.

There are a number of intrinsic relationships between class items in the library. The most important in object oriented programming is the ISA or inheritance hierarchy. Figure 2-2, shows a portion of the Smalltalk hierarchy. Each class is a specialization of another, called its parent or superclass. The only exception is the root of the inheritance tree, here class *Object*. Each class inherits all the methods and instance or class variables from its parent and by transitivity all its ancestor classes.

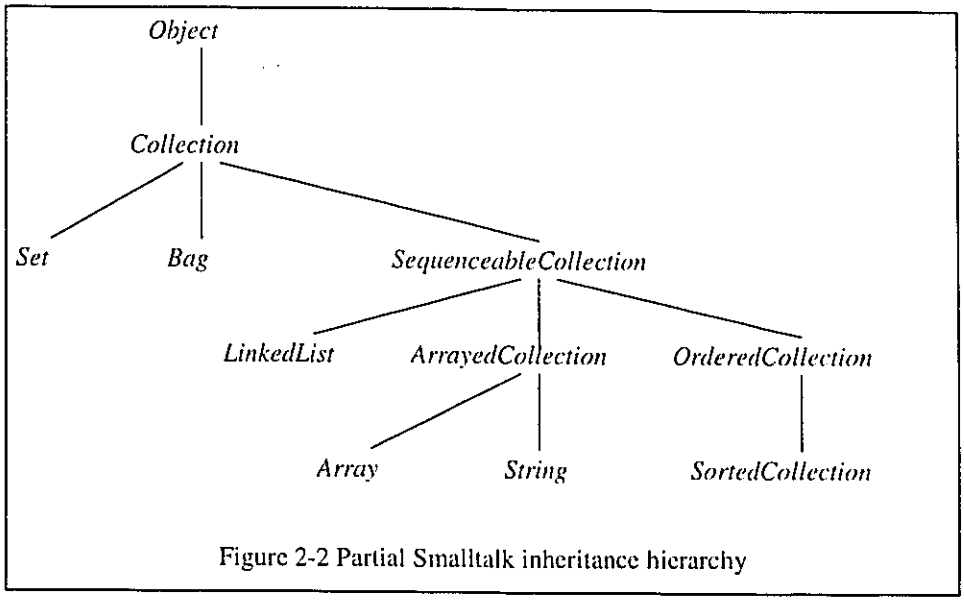

Figure 2-2 Partial Smalltalk inheritance hierarchy

The browsable library is represented internally as an annotated graph. Each class is mapped to a frame-like class node (as in Figure 2-3), where each field represent either a class relationship or a feature of that class. Class features include the class name, instance variables, and class variables. Each class node also contains a set of subnodes where each subnode represents a method the class implements.

Class Node	
name	name of the class
instVars	list of instance variables
classVars	list of class variables
instance method	list of implemented instance methods
class method	list of implemented class methods
superclass	list of superclasses
subclass	list of subclasses

Figure 2-3 Class node schema

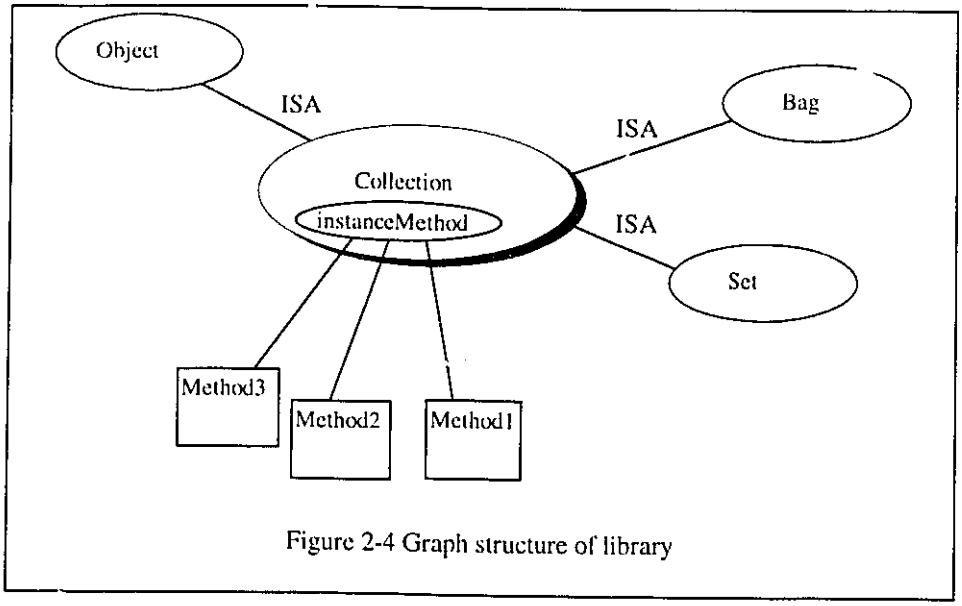

Figure 2-4 Graph structure of library

Figure 2-4 shows the graph representation of a subset of the class hierarchy displayed in Figure 2-2. The ISA links represent the node links, i.e. the superclass and subclass relationships between class nodes. The instanceMethod feature of the Collection class node points to the list of implemented methods, each being represented internally as a method node.

Search through the graph library, triggered by either a browser or active browser query, follows a process similar to spreading activation. All browser queries are mapped into a Guidar [Duchier, 93b] search query. The Guidar search process is based on the spreading computation paradigm [Duchier, 93b] which proceeds by propagation from a set of starting nodes through the graph library, evaluating every node and annotating them with a score. On a second pass, interesting (or highest scoring) nodes are retrieved based on the node's annotation. For efficiency purposes, our search process sidesteps the second pass. Since in our application the annotations are scores, during the first iteration a list of the n best nodes is updated every time a node is evaluated, thus eliminating the need to collect the best nodes in a second iteration.

The simplest query consists of a node link traversal. Browsing operators such as subclass or superclass are such queries. The result of these queries are the nodes adjacent to the "start" node. In the case of a browsing operator, the start node is the node to which the operator was applied. Link traversal queries can also encompass more than one link. In

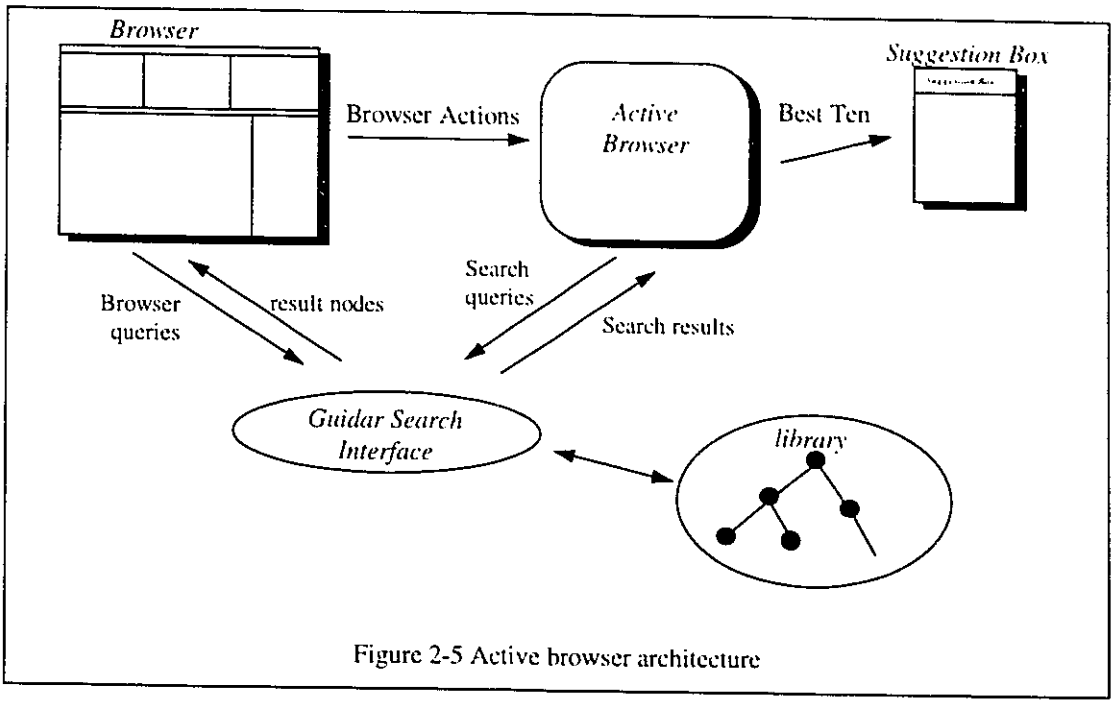

Figure 2-5 Active browser architecture

the class "sibling" operator, for example, search propagates to the parent node and thereafter to the children of that parent node. The link traversal queries follow a navigation pattern defined by the query. More complex type of queries express node relationships that are not described by the static graph links, and involve evaluating nodes based on a pattern. Examples of such queries are "classes using a set of methods" or "classes similar to a class". Complex queries are handled by the spreading computation search.

The architecture of an active browsing system is summarized in Figure 2-5. The main components are identified: the "normal" browser, the active browsing component, the library and the Guidar language interface.

Each browser interacts with the library through the Guidar search language. Guidar search queries are processed on the library and resulting nodes returned for display. For instance, the browser translates user browsing actions into queries, and similarly, the active browser queries the library for nodes that best match a set of specifications (i.e. given a set of class names and method names).

2.2.2 Browser User interface

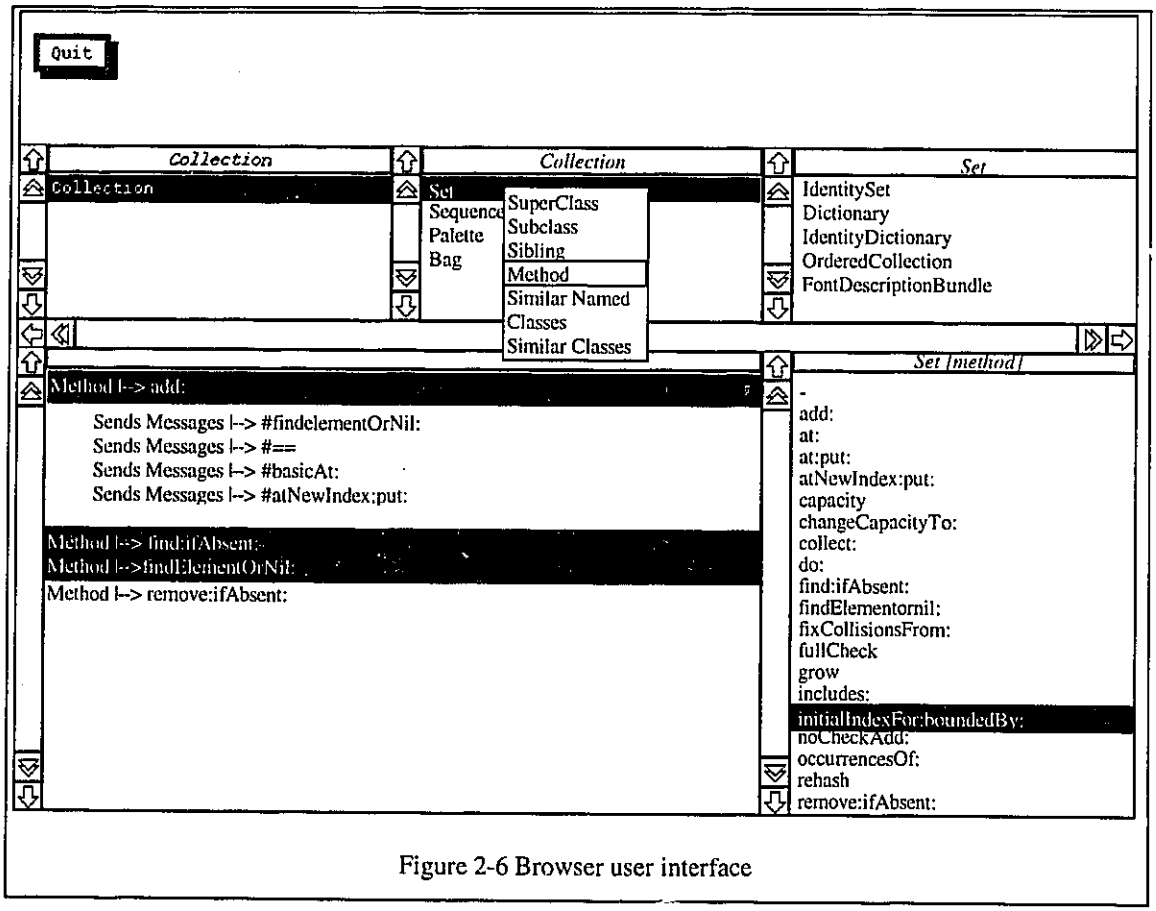

Figure 2-6 Browser user interface

The browser, similar to the Objective-C browser [Drummond, 92], is the user's interface to the class library.

The Browser interface is shown in Figure 2-6. It is divided into three main parts. Each of the top series of windows can display a list of classes. The lower right window is the "method list" window. It displays the methods implemented in a given class. The methods themselves can be expanded in the lower left window, the "method expansion" window, to show greater detail. Within each window, different browsing actions can be selected from a pop-up menu.

From the class lists the user can browse either the inheritance hierarchy or similar named classes, where similarity is a measure of the match between two class names or similar classes (classes with similar interface), where similarity is measured by the match

between method names of two classes. Selecting a class from any class list and applying the browsing operator "Method" results in the display of the methods implemented by that class in the "method list" window. When a method is selected from the method list it is displayed in the bottom left, method expansion window.

In the method expansion window, browsing actions are applied to methods. These actions can be obtained by first highlighting a set of methods and selecting an operator from the pop-up menu. For instance, the user can display methods in two levels of detail. First, expanding a method displays its instance variables, class variables and messages sent. Secondly, the method can be examined in greater detail by displaying its source code.

More interesting operators allow the user to browse class-methods relationships. For example, the "Implemented In Classes" operator returns a list of classes that implement a set of highlighted methods. This operator is useful in that it allows the user to inspect a set of methods, select interesting ones and generate a list of classes that implement the methods deemed interesting. Repeating the same steps by selecting a class from the generated class list and expanding it, the user can systematically get closer to the search target. In the class list generated by the "Implemented In Classes" operator, classes that implement all selected methods get a score of one. Classes that implement only some of the methods or have methods where only some words match with a selected method get a score less than one.

Other operators return a list of classes that is displayed in a class window. These operators take as arguments the set of highlighted methods. The "Used by Class" operator returns the set of classes using the highlighted methods. Similarly, the "Using Class" operator returns the set of classes used by the highlighted methods.

In addition to the normal browser interface, the active browser adds another window to communicate the results of its inference to the user. The suggestion box is a separate window displaying the ten class items that best match the current template. It is updated whenever the analogue changes.

2.3 Inference system

In this section, we describe the inference system, which is at the heart of the active browser. The main purpose of the inference system is to infer the analogue from user browsing actions. It is a standard rule-based system and consists of

- a working memory of facts describing the current analogue and the user's most recent browsing action.
- inference rules that update the analogue upon receipt of new browsing actions.
- a forward chaining inference engine in OPS5 [Brownston et al., 85], that supports conjunctions, disjunction and negation.

2.3.1 Working memory

The inference engine is triggered when a browsing action is received from the browser. The incoming browsing action is added to working memory as a fact and rules for which the antecedent are satisfied are fired. The effect of the rules is to update analogue predicates or create new predicates in working memory.

The browsing action received by the inference engine is a list consisting of the operator, the item it was performed on and other contextual information. This action is translated into a fact in working memory of the form

(Browsing-action node-type node-name operator prior-node prior-op)

The node-type and node-name describe the item the browsing operator was applied to. The prior-node and prior-operator are the name of the node and operator that produced the list from which this node was selected. The operator is the menu selection applied to the node.

The analogue consists of a collection of facts residing in working memory. Each fact is of the general form

***InterestedIn*(node-type node-name confidence-factor)**

This fact expresses the system's belief that the user is interested in a node of type node-type and node-name. The node-type is either "class" or "method" and node-name is respectively a class name or a method name. The confidence-factor is a measure of the degree of confidence of the system in the inference of the predicate. For example the

following analogue predicate indicates that the system believes with 0.75 confidence that the user is interested in a class named Integer.

***InterestedIn*(class Integer 0.75)**

The confidence factor of an analogue predicate varies between zero and one. It is updated as a result of a rule firing as explained later in this section.

2.3.2 Inference engine

The inference engine contains a set of rules that maps user browser actions to the analogue. A rule's antecedent (or LHS) involves user actions and existing facts of the analogue, and its consequent causes new facts to be generated or old ones to be updated. In this way, the analogue is updated to reflect the user's current interest. The general form of the rules is

If ((browsing action) and (existing analogue predicate) ...)

Then ((new analogue predicate) and

***Remove*(browsing action))**

The rules allow the conjunction or disjunction of facts in the LHS of the rule. If no connective is specified conjunction is the default. In addition a pattern in the LHS may be negated by putting a "-" in front of the normal fact. The system only uses positive confidence but facts do allow negation. For the antecedent a negated fact is true if the fact does not exist in the working memory. After a browsing-action fact is consumed by a rule firing, it is discarded from working memory as specified by the OPS5 "action" *Remove*.

2.3.2.1 Inference rules for generating new analogue predicates

Figure 2-7 shows the inference rules related to the analogue (the OPS5 rules can be found in Appendix A and B). Rules one to five refer to class items while Rule6 refers to method items. All rules are triggered by an incoming browsing action which is "Removed" from working memory afterwards (for clarity the rules do not show the remove OPS5 action). A given fact can trigger several rules. To arbitrate between rules that have LHS matching, only the rule with the most facts in its LHS is fired.

```
Rule1: If    (Browsing-action class <node>)
       Then  (InterestedIn class <node> 0.01)
Rule2: If    (Browsing-action class <cur-name> <op> <name> "Subclass")
             (ChildrenOf class <name> <cf>)
             - (InterestedIn class <cur-name>)
       Then  (ChildrenOf class <name> <cf>)
             (AclassInh <name>)
             (InterestedIn class <cur-name> 0.01)
Rule3: If    (Browsing-action class <cur-name> <op> <name> "Subclass")
             - (ChildrenOf class <name>)
       Then  (InterestedIn class <cur-name> 0.01)
             (ChildrenOf class <name> 0.02)
Rule4: If    (Browsing-action class <cur-name> <op> <name> "Subclass")
             (SiblingOf class <pname> <name> <cf>)
       Then  (AclassInh <name>)
             (InterestedIn class <cur-name> 0.01)
             (SiblingOf class <pname> <name> <cf>)
Rule5: If    (Browsing-action class <cur-name> "Subclass"  <name> "Superclass")
             (InterestedIn class <name> <cf>)
       Then  (InterestedIn class <cur-name> 0.01)
             (SiblingOf class <name> <cur-name> 0.01)

Rule6: If    (Browsing-action method <method-name> <op> <name>)
       Then  (InterestedIn method <method-name>  0.01)
             (InterestedIn class <name> 0.005)

Rule7: If    (InterestedIn (class class-name <new-confidence>) new
             (InterestedIn(class class-name <old-confidence> ) old
       Then  (InterestedIn(class class-name <combined-confidence> ) old
             (Remove InterestedIn(class class-name)) new
```

Figure 2-7 Active browser inference rules

The simplest rule, Rule1, creates a new analogue predicate when the user browses a new item. It is based on the assumption that when the user applies a browsing action to a node the information at that node is interesting. When the user applies a browsing operator to a class item <class-name> an analogue predicate is added to working memory to show interest in that class. The confidence factor is set to 0.01.

Any of rules 2 to 5 is triggered when the user applies a browsing operator to a class item. Rules 2,3 and 4 refer to the case where the user's previous action was the "Subclass" operator. Rule2 infers that if the user expands the child of a class for a second

time, the children of that class should be generalized. The first clause in the RHS shows that the user is interested in the children of the parent class. Rule3 is slightly different from Rule2. It is only fired when the user is expanding any of the children of a class for the first time. Since the user has not visited more than one child we do not infer that the children should be generalized. Rule4 reinforces the user's interest in the sibling of a class if the user expands the child of that sibling. Rule5 infers that the user is interested in the sibling of a class when the user visits the children of its parents.

Rule6 is fired whenever the user applies a browsing action on a method item. The rule infers that the user is interested in that method and the class that implements that method. As a result two InterestedIn predicates are added to the analogue, the first one for the method name and the second one for the class name.

2.3.2.2 Combining confidence factors

Whenever a rule generates a fact that already exists in working memory the two facts are merged into one and the confidence factors are combined. Each analogue fact is tagged with a "new" or "old" status. A "new" fact is created as a result of a rule firing. A fact is "old" if it was created previously. When a "new" fact is generated and if a similar fact already exists in working memory, their confidence factor is combined. The "new" fact is discarded and the confidence factor of the "old" one is updated. The combination of confidence factors is achieved by rules similar to Rule7, one for each type of analogue predicate, where the new confidence factor is computed using the following formula:

combined-confidence = old-confidence + (1 - old-confidence) * new-confidence

This has the intuitive effect that confidence is increased proportionally to the difference between certainty and the old confidence. Thus only if the new fact is definitely true is the overall confidence set to one.

The remove action in the RHS discard the "new" analogue fact after its confidence factor is updated. At all times there are at most two similar analogue facts in working memory, a "new" one and an "old" one. After the fact's confidence factor is updated only the "old" one remains.

2.3.2.3 Reinforcing relevance of analogue predicates

The more often the system infers the user's interest in an item the higher the confidence of the system in the relevance of that item. For example in Rule6 each time the user selects an action on a method within a class, the system infers that the user is interested in that method and that class. As the user selects more methods of that class the system infers that the user's interest in that class is higher and increments the confidence factor of the *InterestedIn()* predicate of that class. Through Rule7, the relevance of an item's analogue predicate is reinforced the more the user browses items related to that item. The confidence factor of analogue predicates for methods are reinforced for different reasons. Due to polymorphism in OO languages different classes may implement methods bearing the same name. As well the inheritance hierarchy implies a specialization of classes. A class may reimplement a method of its parent to "specialize" that method for itself. By browsing different classes but similar named methods the user reinforces the system's confidence that these methods are interesting.

However irrelevant items encountered during browsing produce "noise" in the analogue; i.e. analogue facts that are not relevant to the search target. When the frequency of "visiting" irrelevant items is low the noise in the analogue is not significant. This assumes that the user's browsing search is focused and rarely steers away from the original goal, and the user "hits" relevant items with high frequency.

The amount of noise in the analogue can be too high in several cases:

- Usually at the beginning of a browsing session the user has only a vague idea of the requirements describing the search target.
- Sometimes the search target may be a class that is unique in the library and bears little similarity to any other class in the library. During browsing the user may seldom find classes relevant to the search target.

In such cases the user selects items at random and hits many items irrelevant to the search target.

Noise is thus always present in the analogue but active browsing infers the search target because of the relevance of reinforced predicates to the search target. The quality of the analogue, and hence the inference performance of active browsing, is thus a result of

the user's browsing process. The closer the user is to the search target, the better the analogue matches the search target. On the other hand, the active browser may not be able to infer the search target when the user rarely selects relevant items.

2.4 Template Matching and scoring

The analogue represents internally the user's search target as inferred by the active browser. Analogue predicates describe class features that the system believes to be interesting to varying degrees, measured by the predicate confidence factor. In other words, the analogue is a set of specifications representing the search target.

The objective of the active browser is to display items that are "interesting" to the user. While the analogue is the system's inference of the user search target, the "template" is the mechanism whereby through matching items most relevant to the user are retrieved from the library.

After inferring the analogue the system needs to translate this set of specifications into a concrete list of library items that satisfy these specifications. The list is ranked according to their relevance to the specifications, the highest ranked item being the "most interesting" to the user. This is achieved by converting the analogue into the template that can be readily used to measure the relevance of a library item to the user. "Matching" is the process of evaluating the library items on the template and ranking them according to their score.

Inferring the template is comparable to formulating a query. The matching process which retrieves items that best match the template is similar to evaluating that query.

2.4.1 Template

Analogue Predicates ==> Template Predicates ==> Template Schema

As with analogue predicates, template predicates are generated by inference rules and converted into a frame-like structure similar to a class node.

The template represents the analogue in a form suitable for matching. The form of the template is determined by the matching process and is not necessarily a direct mapping of the analogue. For example template predicates can be used to reflect changes in

```
TemplateSchema
        className = listof ( ("Array" .5) ("ArrayedCollection" .7) ... )
        impMeths  = listof ( ("at:put:" .66) ("at" .46) ... )
                  :
```

Figure 2-8 Template schema

successive states of the analogue, in which case each node must contain its score. During matching the score of each node is incremented. The added benefit of this method is to speed up the scoring process as the same class is not evaluated on previous template terms at every single update of the library.

The template schema contains fields similar to a class node (see Figure 2-3). Each type of analogue predicate maps uniquely to a template field. However, each field contains a list of <value,weight> pairs, where weight is derived from the analogue and carries a value between 0 and 1. Figure 2-8 shows a template schema with the two main fields containing class names and method names (other fields include instance variables, class variables and exclude class). The term "ArrayedCollection" carries a higher weight than "Array", which means that the system's confidence in the former is stronger.

In [Drummond, 92] template predicates were generated within the rules that updates the analogue. In this thesis we have decoupled template generating rules from rules that update the analogue. The system responsible for inferring the analogue is thus independent of the matching process. As a result the format of the template does not impact the design of the inference rules. The inference engine can be more easily ported to different repositories which may require a different template format.

2.4.2 Matching and scoring

Matching is the process of scoring every class from the library on the template and returning the n highest scored class nodes. The graph library is searched using spreading computation and each class node is evaluated on the template when visited. The template schema being structurally similar to a class node schema, a class node is scored by evaluating the corresponding fields in the template: i.e. className is scored on every <className, weight> pair in the class name field of the template.

```
score = 0
for each template term ( value,w ) pair do
        case type ( template term ) of
                class :    score = score + score_class_name (className, value) * w
                method : score = score + score_method_name ( methodName, value ) * w
```

Figure 2-9 Template scoring algorithm of class names

Figure 2-9 Template scoring algorithm of class names illustrates the process of evaluating the score of a class on a template schema. For each template term, the score of the class is incremented by the product of its score on that term and the weight. The two main template terms, class names and method names are evaluated differently. The class name is evaluated on a template class type term while a template method type is scored on all of the class methods. The first is handled by the *score_class_name* function and the latter by the *score_method_name* function.

2.4.2.1 Class name scoring

Class names are made up of words, e.g. "BaseLayer" and "ArrayedCollection".

```
; function calculating the similarity measure between a
; library class name and a template class name
function match_class_name ( lib_classname, templ_classname )
begin
   split lib_classname into words
   for each word in lib_classname
       if word in templ_classname then
          pos := position of word in lib_classname
          weight := 1/pos
          if pos = position of word in templ_classname then
              score := score + weight
          else
              score := score
                    + weight / (1 + abs ( pos - position in templ_classname ))
          endif
       endif
   endfor
   return score
end
```

$$\text{normalized score} := \frac{match\_class\_name(lib\_classname, templ\_classname)}{match\_class\_name(templ\_classname, templ\_classname)}$$

Figure 2-10 Pseudocode of class name matching algorithm

The similarity measure compares a library class name against a template class name by their constituent words. The score is biased on the position of the words in the class name. The position is counted from the last word in the class name, thus the last word has a position of one.

The two class names are compared word by word. Each word in a class name carries a weight inversely proportional to its position; the last word would have a weight of 1 and the second to last a weight of 0.5. For each word in the library class name the score is incremented as follows. If the word occurs in the template class name and the position of the word in both correspond, the score is incremented by its *weight* $= 1/position$. If the word occurs in the template class name but at a different position, the score is incremented by the weight reduced by the difference in position, i.e.

weight / 1+ abs(position_in_template_class_name - position_in_library_class_name).

To normalize the similarity measure the score is divided by the score of the template class name measured on itself. Figure 2-10 shows the pseudocode of the class name matching algorithm.

2.4.2.2 Method name scoring

Method names also consists of multiple words, but they are scored differently. The bias is placed on the first word as this is the most informative term in method names. Also, method name scoring takes into account inheritance. In object oriented languages, a class inherits the methods of its superclass and by transitivity methods of its ancestors. A class may not implement a method but any method it inherits can be invoked on the class, and is part of the behavior of that class. In our scoring scheme a class is given credit if its ancestors implement the method the class is scored on.

The *score* of a class on a method name *m* is calculated by the following formula

$$score = 0.7 * impl\_score + 0.3 * super\_score$$

where *impl_score* is the score of the class based on the similarity of the methods it implements to *m*, and *super_score* is the *score* of the superclass on method *m*.

The *impl_score* is given by the similarity measure for method names. If the method is implemented in the class, the impl_score returns a value of 1. Otherwise, *impl_score*

returns a score less or equal to 0.8 as follows. First, methods implemented in the class are split into words to make a set of words for the class evaluated. Then the template method name is split into words and starting from the first one, if it exists in the class's set of words, the score of the class is incremented. However, the score on the first word is incremented by 0.66 and from the second word onwards, the increment is

(0.8-0.66)/number_of_remaining_words.

This algorithm ensures that an exact match returns 1, a partial match (in which words might be in the wrong position or occur in different methods of the class) that includes the first subterm returns a maximum value of 0.8, and other partial matches return at most 0.66. Note that contrary to [Drummond, 92] the position of the words in the class's methods is not a factor in the score. To do so would require a more complex algorithm and is not taken into consideration for efficiency reasons.

2.5 Active browser performance

In this section, we illustrate the efficiency of the active browser with an example. We measure performance by comparing the ranking of a specified target in the suggestion box against the ranking of the user search target. The ranking of the user is given by the ranking of the target in the browser's most recent class list.

Table 2-1 displays the sequence of actions taken by the user searching for the class "PopUpMenu". The user first chooses class "Depth8Image" from the initial class list and applies the "Method" operator to it. The "Method" operator displays the methods implemented by the class "Depth8Image" in the "method list" window. Some of the methods are inspected and marked as interesting. The user then applies the "Implemented In Class" operator to the marked methods. As a result a new class list is generated, which corresponds to a step in the user's search. From this class list the user inspect three classes before marking interesting methods in the third class "Screen". Here again the "Implemented In Class" operator is applied on the marked methods and the third class list generated. In subsequent steps the same pattern of search is followed.

step	Class or Method	Operator
1	Depth8Image (marked methods)	Method Implemented In Class
2	Depth2Image Depth16Image Screen - (marked methods)	Method, inspect methods Method, inspect methods Method (mark method) Implemented In Class
3	ScheduledWindow - (marked methods)	Method (mark methods) Implemented In Class
4	Window - (marked methods)	Method (mark methods) Implemented In Class
5	DisplaySurface - (marked methods)	Method (mark methods) Implemented In Class
6	FileConnection - (marked methods)	Method (mark methods) Implemented In Class

Table 2-1 Sequence of Browsing Actions

Table 2-2 shows the rankings of the search target, class "PopUpMenu", in the class lists of the active browser (suggestion box) and the user. The first 6 steps are shown. From step 2 onwards, the target appears in the suggestion box. This example clearly demonstrates the convergence produced by the inference system (from step 1 to step 6, ranking improves monotonically in the suggestion box) whereas the user is wandering away from the target (at step 4 the user gets closer but then diverges after that). Every improvement in the ranking of the active browser, between consecutive steps, indicates that the analogue is being reinforced by the user's browsing actions.

steps	active browser	user
1	12	95
2	10	95
3	9	57
4	8	3
5	8	16
6	6	11

Table 2-2 Class lists rankings for search on class "PopUpMenu"

This brief discussion is meant to be illustrative only. We defer the discussion of the active browser performance to later chapters.

2.6 Conclusion

This chapter explained the method of active browsing and the concepts behind it. The system outlined, which will be referred as the Base system in subsequent chapters, differs in several respects from the original active browsing system [Drummond, 92]. More specifically the inference rules have been redesigned to separate the template generating rules from the rules that update the analogue, and the method scoring algorithm has been simplified for efficiency purposes. This thesis is about improving the performance of the Base system. Chapters 4 and 5 describe two methods of achieving this. The next chapter describe the method of validation used and the performance obtained in the Base system.

Chapter 3

3. Experimental Method

The methods developed in this thesis were evaluated empirically. An active browser with negative inference, "negative active browser", and an active browser with selective search, "selective active browser", were compared to one another, to the Base system described in the previous chapter and to normal browsing.

This chapter presents the experimental method which is the same as in [Drummond, 92]. We also use an automated search agent, Rover, introduced in [Drummond, 92] and described in section 3.2. Section 3.3 describes briefly the experiments conducted and section 3.4 describes the data collected and the various methods of comparing the output of the different active browsers.

3.1 Experimental method

The basic method of evaluating the performance of an active browser is to monitor the search for a target in a library by a search agent (human or automated) and compare the number of steps to reach the target with and without the active browser. The result of a search is a win for the active browser if it finds the target before the search agent, a loss if the agent finds the target before the active browser, and a draw otherwise. The number of wins for a sample of targets is our primary measure of an active browser's inference accuracy.

In designing the experiments the following characteristics were considered desirable:

- A large sample of searches: in order provide an accurate average measure of an active browser's performance.
- Repeatability and consistency: the search for a target must be easily repeatable so as to evaluate various active browsers on the same target. Furthermore, the search agent must be consistent through each session as the results are used to compare active browsers.

- Human users: active browsing is designed to help software engineers browsing a library they are unfamiliar with. The ideal subjects for these experiments should have average browsing skills and little familiarity with the library.

Using human subjects involves some constraints on the experimental setting that make large scale experiments unfeasible. Humans would perform inconsistently over time: humans learn with experience: the more one interacts with the browser, the more proficient one becomes at searching for items in the library. Experiments cannot be repeated with the same user, finding a class a second time would be trivial. Moreover, the variability of user's browsing skills and familiarity with the search domain means that a large sample of users with varying skill level is required.

Due to the impracticality of running large scale and controlled experiments with human subjects, an automated search agent is used instead. Experiments can be repeated consistently with an automated search agent. The same agent can be made to search on the same target twice, with the same result. An average measure can be obtained using a large sample of searches, by setting each class in the search library as search target.

3.2 Rover an automated search agent

A search is controlled by the user's internal representation of the goal. In a typical browsing session, the user selects an item and evaluates its relevance to his or her specifications of the search goal. Based on the result of this evaluation, the user selects a browsing operation which he/she believes will bring him closer to the goal. The decision making process is fuzzy as more often than not, the search goal is not fully understood or clear.

Rover is an automated search agent playing the role normally assumed by a human user. Rover's search process includes the same fuzzy characteristics as a human user.

3.2.1 Components of Rover

Rover consists of two main components, an oracle representing the search goal and relevance evaluation and the heuristic agent that maps this to browser actions.

The oracle contains the specifications of the target class (the name of the class and its implemented methods) and answers three specific questions. Asked if a class name matches the target name, the oracle returns a value between 0 (no match at all) and 1 (exact match), an intermediate value meaning that the class names match partially. The oracle returns true or false, when asked whether a method name matches any in the goal. Finally, it returns a value between 0 and 1, when asked whether a set of method names is a better match than a previous one.

To simulate the uncertainty of human users, the oracle sometimes returns incorrect answers. On the first question and last question, the oracle's answer is fuzzy, but on the second question, it always returns the correct answer. When asked whether a class name matches the target, 30% of the time the oracle answers yes. Of the remaining 70%, the oracle answers no only if the class name does not match the target class at all, and answers yes otherwise. On the third question, when asked whether a set of methods matches better to the target than a second set, it returns true if the first set is a better match than the second. In the case where the second set is a better match than the first it answers "no" only 75% of the time.

The heuristic search agent uses a subset of the available standard browser operators and implements a single search strategy that selects appropriate browsing operators according to the answers of the oracle. The search algorithm is described next.

3.2.2 Rover Search Algorithm

Figure 3-1 represents in pseudocode Rover's search algorithm. The search starts from an initial alphabetical list of classes. Rover selects the first class. It then queries the oracle whether this class name is similar to the target class. A match above a threshold value results in the expansion of that class to show its implemented methods. Otherwise, Rover selects the next class.

When Rover searches a class list other than the initial list Rover visits up to a maximum of ten classes until it finds a good matching class, backtracking to the previous class list if that list is not the initial one. When Rover finds a matching class it expands the class to list all of that class's implemented methods.

```
1. Select next class name in class list
        IF class name identical to target STOP
        ELSE IF class name not in top ten of list
                THEN backtrack to previous list
                        repeat from 1
        ELSE IF class name does not match with target
                THEN repeat from 1
                ELSE go to step 2
2. Expand class
        assign maximum number of methods from 3 to 5
3. Select method at random
        IF method does not match with target
                THEN repeat from 3
4. Save method
        IF number of saved methods less than maximum
                THEN repeat from 3
5. Score saved methods against target class
        IF score less than best previous class
                THEN repeat from 1
6. Apply "Implemented in class" operator to saved methods
7. Repeat from 1
```

Figure 3-1 Rover Algorithm

At step 3, from the list of methods Rover selects a method name at random and asks the oracle if it matches with methods of the target class. If the answer is "yes", that method is saved and Rover repeats step 3. This continues until a maximum number of methods have been saved or the method list is exhausted. Rover then asks the oracle if the set of saved methods is a better match to the implemented methods of the target than those selected in the best of all previously opened classes. If the answer is yes, the operator "Implemented in Classes" is applied to the set of saved methods and the new generated class list becomes the current class list that Rover browses from. Otherwise, Rover rejects the expanded class and continues browsing through the class list.

Rover repeats this process in the most recent class list until any of three conditions is true. If Rover finds the target, the search stops. If a class better - according to the oracle - than the last visited one is opened, a new class list is generated and the cycle is repeated. If the search reaches the tenth class in the current class list then Rover backtracks to the previous list and continues the search from the next class in that list.

3.2.3 Differences with original automated search agent

The original automated search agent [Drummond, 92] is rule based and used the same inference engine as the active browser. Our reimplementation of Rover's algorithm is written in Lisp in functional language style. Aside from a different choice of implementation language other differences exist between the original search agent and Rover.

The answers returned by the oracle are different in two ways. First, to the second question relative to whether a method name match any in the target class, our oracle's answer is not fuzzy, it is always correct. Preliminary testing of Rover showed that with noise on the second question Rover performs poorly, its search not converging to the target. Removing noise produces a search agent capable of finding the search target in reasonable time.

The method of adding noise to the oracle's answers is also different from that in the original search agent. In [Drummond, 92] the noise factor is a uniform random variable whose mean and variance can be set within the search agent's rules. If the match value is above a threshold determined by the random variable a yes is returned otherwise a no. This scheme allowed easy setting of different noise factors to study the search agent's performance with respect to noise levels.

3.2.4 Ranking list of an experimental run

One experimental run consists of selecting a class as the search target and starting Rover. During Rover's search browsing actions are generated and the active browser updates the suggestion box by matching all classes in the library against its inferred analogue. The result of this matching is an ordered list of library items of which only the first ten are displayed in the suggestion box. The "ranking" of the target is the position of the target in the ordered list of items after the library is evaluated; a ranking of one is best. The ranking is a measure of the accuracy of the analogue, hence the inference accuracy of the active browser. The higher the position of the target in the list of evaluated items, the better the active browser.

The "ranking list" of one experimental run is a list in which the target's ranking at every step of the run is recorded. A step in a run is when Rover generates a new class list or backtracks to a previous one. The ranking list shows the change in the ranking of the target by the active browser as search progresses. The ranking list is a useful tool for comparing active browser inference performance.

3.3 Experiments

The basic Smalltalk source library of 389 classes was used as the test library. Each class was set as the search target in turn. Rover's search is started and the ranking lists of both the active browser and Rover recorded. The search is stopped for any of the following reasons. Rover finds the target. The active browser "identifies" the target. The search is too long: for all practical purposes neither Rover nor the active browser could find the target in a reasonable number of steps. A run is too long when neither Rover nor the active browser find the target after 70 steps.

A run is considered invalid when the search is either too long or too short, where the length of a search is the number of steps before search stops. A run is considered too short when the search is shorter than the number of steps required to award a win to the active browser (when the target is displayed for five consecutive steps in the suggestion box). Of the 389 runs, there are a number of invalid runs and four classes that cannot be used as targets because they have no implemented methods.

3.4 Comparing and evaluating searches

The ranking lists of both Rover and active browser is the primary data used for evaluating the active browser performance and comparing one version to another or to Rover. We used two different methods of comparing performance. The first one consists of comparing the number of "wins" of the active browser over Rover. The second method compares at each step the ranking of the target in each active browser and the number of active searches.

3.4.1 Assigning wins

Each search is declared either a win, a loss or a draw for the active browser, a win indicating that the active browser correctly inferred the target before Rover. A win is awarded to the active browser only after the target class is in the suggestion box (ranking 10 or higher) for five consecutive steps. The active browser is assigned a loss when Rover finds the target and its rank is higher than that of the active browser. In all other cases, the search is considered a draw.

To evaluate an active browser with respect to Rover, we counted the number of searches where the active browser wins, loses or draws with Rover. The different active browsers can thus be compared by their win/loss statistic, the better active browser having higher a wins count and/or a lower losses count.

3.4.2 Comparing active browsers

The win statistic is a useful metric for comparing active browsers. However active browsers can also be compared using the ranking of the target.

A "winning" run is an indication of the speed performance of an active browser, i.e. that it can find the target faster than Rover. However merely counting a run as a win or loss ignores the ranking of the target. The ranking of the target is a direct measure of the system's accuracy at inferring the target, which we refer to as inference quality.

Also, a win evaluates a search on the last five steps irrespective of how good or bad has the target ranked before the last five steps. Target ranking at every step of the search is a measure of the inference performance of the active browser at any step during the search. This information is more useful than wins when comparing active browsers. Clearly the better active browser is the one which ranks the target consistently higher throughout the search.

Ranking information can also help determine the "better" active browser among the winners. On a given target T, two active browsers AB_1 and AB_2 may win when running against Rover. The search for T on both would be counted as a win, although the position of the target in the suggestion box may be higher in one than the other (although

the difference cannot be greater than 10). Clearly, the better active browser is the one in which the target is positioned higher in the suggestion box.

In our study, the rankings for all runs are averaged out and displayed in graphical format. Prior to the discussion of average target ranking, we need to introduce the concept of active search.

3.4.2.1 Active search graph

A search is active until the target has been identified. For Rover, search ends either when it finds the target or when search length reaches the maximum limit. In the case of the active browser, search ends when either it wins, loses or draws with Rover.

The active search graph displays the number of searches still active at each step. The x-axis is the step number in a search. The y-axis represent the number of searches still active at step x. For practical purposes, the maximum search length has been set to 70.

When comparing the active search graph of two active browsers, a lower y-value means a better active browser. A lower value indicates that more targets have been identified at or before this step. The vertical difference between two plots at step x, reveals the number of searches that have ended in one system but are still active in the other.

In practice, when Rover finds the target before the active browser search ends for both. The search is not "active" anymore for both Rover and active browser even when Rover wins. On the other hand, when the active browser infers the search target before Rover the search in Rover is still "active" afterwards. For this reason the number of active searches at any step is always less for an active browser than for Rover. In the active search graph all active browser plots are always below that of Rover's. However this does not apply when comparing active browsers.

3.4.2.2 Average Ranking graph

The average ranking graph plots the average ranking of the target at each step of a search. The average rank at step x is the ratio of the sum of the target ranking of all active searches to the number of searches active at step x. The larger the average ranking the worse the active browser.

To compare active browsers, the average ranking curve for each version is plotted on the same graph, the lower the curve the better active browser. The vertical difference between two curves, indicates on average, the difference in the ranking of the target between the respective active browsers.

3.5 An alternative to Rover

3.5.1 Different search strategy

Rover's search strategy relies heavily on the class interface - i.e. the class's implemented methods - and method relationships. The only browsing operators it uses are centered around methods such as displaying methods of a class and displaying classes that implement a given set of methods. It does not make use of browsing operators based on class relationships and does not exploit the class hierarchy.

However, much information can be obtained from the class hierarchy that may be particularly relevant during browsing. For example, class inheritance is generally used for specializing or refining a class specification, and implies that some commonalties exist between a parent class and its subclasses.

Our alternate search agent (Rover2) explores the class hierarchy. Rover2 uses the "subclass" browsing action during search.

```
1. Select next class name in class list
        IF class name identical to target STOP
        ELSE IF class name not in top ten of list
                THEN backtrack to last list
                        repeat from 1
        ELSE IF class name does not match with target
                THEN IF subclasses match with target
                        THEN newclasslist = subclasses
                        repeat from 1
                ELSE go to step 2
2. Expand class
        assign maximum number of methods from 3 to 5
3 etc ...
```

Figure 3-2 Rover2 search algorithm

Figure 3-2 illustrates, in bold character, the difference between Rover2's and Rover's. A new option is added to Rover's search before displaying a class's implemented methods. At step 1, Rover decides to "open" a class based on whether the class matches the target (an interesting class for Rover) or not. Instead, in Rover2, if the selected class is not interesting, we add the option of generating a new class list if any of its subclasses match the target.

3.5.2 Comparing Rover and Rover2

	Rover	Rover2
average length	12.9	15.5
no. of long searches	18	16

Table 3-1 Rover and Rover2 search performance

First, the performance of both Rover and Rover2 as searching agents were compared. Rover and Rover2 were set to search for every class in the library and a ranking list for each target was generated. Contrary to the active browser's runs, search stops when either Rover or Rover2 finds the target or the search is too long (i.e. longer than 70 steps).

The results for both Rover and Rover2 are summarized in Table 3-1. For completed searches, in which Rover finds the target in less than 70 steps, the average is calculated. On average, Rover2 finds the target more than two steps after Rover. This shows the different strategy clearly affect the performance of Rover. However, the difference is not too significant. This is reflected, in Table 3-2, in the number of searches in which one Rover is faster than Rover2.

	No. of searches
Rover faster than Rover2	176
Rover2 faster than Rover	159

Table 3-2 Comparing speeds of Rover and Rover2

In 176 searches, Rover completes the search before Rover2 while the converse is true in 159 searches. Rover is a marginally better search agent than Rover2.

3.5.3 Active browsers results on Rover2

	Win	Loss	Draw	No. of invalid runs
Rover	161	110	35	83
Rover2	87	193	36	73

Table 3-3 Rover v/s Rover2 wins and losses

The negative active browser was tested with both Rover and Rover2. Table 3-3 compares the win and loss numbers of Rover and Rover2. When tested on Rover, the active browser performs better than with Rover2. The number of wins with Rover is roughly twice and the losses about half that of Rover2.

One can argue that the strategy used by Rover favors our inference rules. The addition of a class browsing action in Rover2 results in Rover2 selecting fewer method items than Rover. As well the search strategy of Rover2 produces more class name features in the analogue. The better active browsing performance would then suggest that method name features is more beneficial than class name features in the analogue. This may be a result of a characteristic of the library. In the Smalltalk language there is a high similarity coupling between method names of different classes due to method naming convention and polymorphism.

For the purpose of comparing active browsers Rover is more adequate than Rover2. When evaluating other versions of the active browser we expect, in some cases, wins to be less than that of the negative active browser. The low win count with Rover2 leaves little room for the other active browsers and thus would make comparing them harder. Rover provides a larger sample of targets on which active browser can win. This allows us to more easily compare versions of the active browser that performs worse than the negative active browser.

The experimental results are presented later in this thesis. Chapter 4 includes the results and conclusion of the evaluation of the negative active browser, and Chapter 6 presents results of the selective active browser.

Chapter 4

4. Negative Inference

4.1 Introduction

This chapter presents the method of negative inference, whose general aim is to improve the inference quality of active browsing. Section 4.2 describes the motivation behind the choice of using negative inference. Section 4.3 presents the method and our implementation of negative inference. Section 4.4 presents empirical evidence of the effectiveness of negative inference. The last section summarizes the chapter and discusses some issues related to the method. The general method of negative inference and empirical results presented in this chapter have been reported in [Holte and Ng, 96].

4.2 Motivation

4.2.1 Shortcoming of inferring method names

In Smalltalk, and object oriented languages in general, method names consist of several words, subterms, concatenated together. For example, in class OrderedCollection, the method *"add:afterIndex:"* , contains subterms *"add"* , *"after"* and *"index"*, and as well, the name of the class is made up of subterms *"Ordered"* and *"Collection"*.

The method of active browsing involves inferring the user's goal, represented internally by the analogue. The analogue consists of a list of features with an associated numerical measure of the system's confidence of the user's interest. Each feature can be either a class, variable or method name. A feature is added to or updated in the analogue when the system infers the user's interest in a particular feature based on his/her browsing actions. For instance, whenever the user inspects a method item displayed in the browser, that method name is added to the analogue

The fact that method names contain subterms makes such inference incomplete. The user may have inspected a particular method because one or more subterms in the

method's name was interesting. For example, when the user inspects the method "remove:index:", three possible inferences can be made: the user may be interested either in "remove" methods, or in methods containing "index", or in methods that contain both subterms "remove" and "index". In the original active browser, the system would infer that the user is interested in the method "remove: index:", although it is assumed that the first subterm is the most significant one and carries more weight. This assumption is incorrect in the cases where subterms other than the first one (e.g. "index") motivated the user's choice to inspect that method. Thus, the analogue does not necessarily reflect the user's intent.

4.2.2 Inferring method subterms

To overcome this problem, the system should be capable of inferring subterms, and the improved analogue should contain subterm features as well. Clearly, given the many possible inferences from a single browsing action, making the correct inference is not a trivial task. The inference system needs to identify the particular subterms the user is interested in.

From the browser point of view, method names are atomic entities and browsing actions are applied to method items but not subterms. When the user selects a method item, the problem for the inference system is twofold:

- decide if the user is interested in the whole name or not
- if not, which subterms are the most relevant ones.

In our method, this effect is achieved by using negative inference, i.e. by inferring what features (subterms) the user is not interested in.

4.3 Positive Inference and Negative inference

The original active browser [Drummond, 92] contained only positive inference rules, i.e. rules that inferred that a user was interested in items having a particular feature. Positive inference is based on the assumption that the user will examine items whose features are similar to the current search goal.

By extension, one could characterize actions available to the user as "positive actions". Other browsers have actions that directly indicate that the user is not interested in a particular item [Lang, 95; Sheth and Maes, 93]. In such systems, negative inference is as straightforward as positive inference. But in our browser, there are no "negative actions", i.e. browsing actions that directly indicate the user is not interested in an item. Negative inference can be based only on actions which could have been taken but were not. Negative features are features that the system believes are not part of the user's search target.

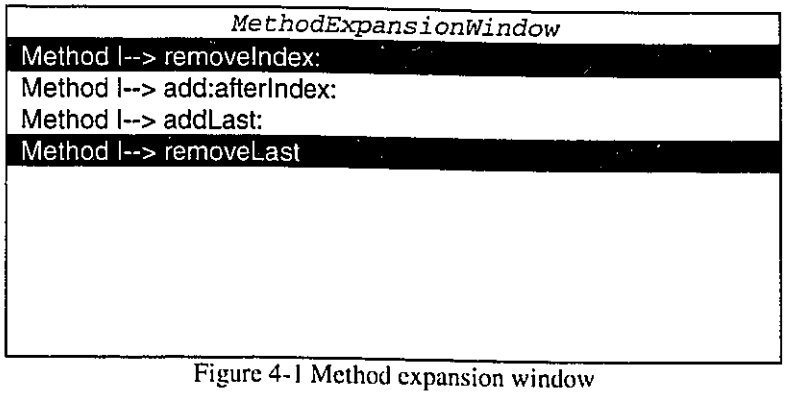

Figure 4-1 Method expansion window

Figure 4-1 displays the view from the method expansion window, after the user opened four methods on the OrderedCollection class. The two highlighted methods, "removeIndex:" and "removeLast", have been marked by the user for subsequent use. From the "marking" action, the system infers the following:

InterestedIn(method, removeIndex:)

InterestedIn(method, removeLast)

Positive inference is based on the following principle: given a list of items X, Y and Z, if the user selects item X, then the system infers that X is interesting. One can naively assume that the user is not interested in Y and Z. However this assumption may not be true in all cases. For instance, the user may have scanned only part of the list and overlooked the rest of the list . To infer that the user is not interested in the rest of the list is incorrect. The system needs to know, with reasonable confidence, that the user considered and rejected Y and Z. Thus Negative inference is not inferring that the user is uninterested in Y and Z, when the user selects X.

4.3.1 Requirements for reliable Negative inference

Clearly at any point during browsing, there are many possible actions of which the user chose only one. It is certainly incorrect to infer that the user is not interested in all the items he did not select. For instance, when the user selects the class OrderedCollection from a class list, we cannot infer reliably that the user is not interested in the other classes in the list. Therefore, to make reliable negative inference, the system needs to identify with reasonably high confidence, which particular item(s) the user consciously ignored.

To infer that an item, say Y, is NOT interesting with a high degree of confidence, two conditions are necessary:

- first that the user considered item Y,
- second that item Y is not selected for subsequent actions.

This is a two step process, made necessary by the lack of negative actions in our browser.

Figure 4-1 illustrates such a scenario. The user has opened the list of methods of the class OrderedCollection, and selected four methods, with the intent of inspecting them in greater detail, in the MethodExpansionWindow. Of the four methods displayed, two are "marked" (highlighted) by the user: "removeIndex:" and "removeLast". These four methods fulfill the first condition, in that by bringing them for display in that window, the user made a conscious decision that these methods may be interesting.

Among the available actions in the expansion window are the "Implemented In" and "Used by" browser actions. The former returns a class list of items that implement the "marked" methods and the latter returns a list of classes that use the "marked " methods. When the user applies the "Implemented In" action, indicating that the user is interested in class items implementing the "marked" methods, then and only then, can one infer with a high degree of confidence that the user deemed the "unmarked" methods to be NOT interesting.

The validity of negative inference is strongly dependent on the type of browsing action selected in step two. The browsing action must be such that one can ascertain, with a high degree confidence, that the user evaluated all items in the list and selected the ones that are relevant to his/her search target. In such a case, inferring the unmarked items as negative features would be correct.

4.3.2 Inferring interesting subterms with negative inference

As presented in section 4.2.3, subterms should be represented in the analogue, but to do this, the inference system needs to identify which subterms of a method name are relevant to the user's search target.

In the preceding section, we presented a general method of reliably inferring negative features in our browser. But a negative method term also indicates that the user is not interested in any subterms of that method term. Thus the negative inference method described above, not only infers negative method name features, but as well, negative subterms. The inferred negative subterms, can thus be used to identify the irrelevant subterms in the positive analogue terms.

This process of subterm pruning using negative inference is triggered by the user selecting the "Implemented In" or "Used by" browsing actions in the method expansion window. The resultant analogue has positive terms in which method subterms' confidence factors are inferred. From the perspective of matching the resulting template, the weight on different subterms of a method in the template is not fixed as in the original active browser, but is inferred. Thus the addition of subterms to the analogue in conjunction with appropriate negative inference produces a more precise analogue.

4.3.3 Negative inference method

Negative inference adds inference rules for selectively inferring both positive and negative features of method subterms. The new analogue is generated in two steps as follows:

1. Augmentation:
    - marked items are added to analogue as positive features
    - marked method subterms are added to the analogue as positive features
    - unmarked method subterms are added to analogue as negative features
2. Pruning:
    - +ve subterm features are pruned from analogue if a corresponding -ve subterm feature is present in the analogue

Positive features are added to the analogue as *InterestedIn(method:<item_name>)* predicates for method names, and *InterestedIn(subterm:<item_name>)* predicates for subterms. Negative features are represented by *NotInterestedIn(subterm:<item_name>)* predicates. Each positive predicate carry a confidence factor as described in chapter 2.

Working memory contains the set of predicates (positive and negative features) that describe the analogue. Let W_i be the working memory at iteration *i* (for this discussion, we will use the term iteration *i* to refer to the *i*th update of the working memory), and W_{i+1} the working memory after the pruning step. Now, the analogue contains both positive and negative predicates, so

$$W_i = W^+_i + W^-_i$$

where W^+_i is the set of all positive predicates in working memory at iteration i, and W^-_i the set of all negative predicates in working memory at iteration i. Similarly, $W_{i+1} = W^+_{i+1} + W^-_{i+1}$.

In step one, when the user marked a few methods out of a list and then selects the "Implemented In" or "Used By" browser command, a set of new predicates are generated which we will call N.

$$N = M + S_u + S_m$$

Where M is the set of method name predicates for each marked method. The set of method names displayed is divided into two mutually exclusive sets: the set of marked methods and the set of unmarked methods. S_m is the set of all subterm predicates in marked method names, and S_u the set of subterm predicates of the unmarked methods.

Then the augmented working memory $W^a_i = W_i + N$, contains the newly generated predicates (= N), both positive and negative.

In step two, pruning is performed. Positive predicates in the augmented working memory are removed if there exists a corresponding negative one. Two types of pruning occur at this stage. First the set S_m of new positive subterms predicates is pruned against $W^-_i + S_u$, the set of all negative predicates in the augmented working memory. Secondly W^+_i , the set of old positive predicates is pruned against new negative predicates i.e. S_u. Thus the resulting working memory after iteration i contains:

$$W^+_{i+1} = (S_m - (W^-_i + S_u)) + (W^+_i - S_u) + M \text{ and,}$$

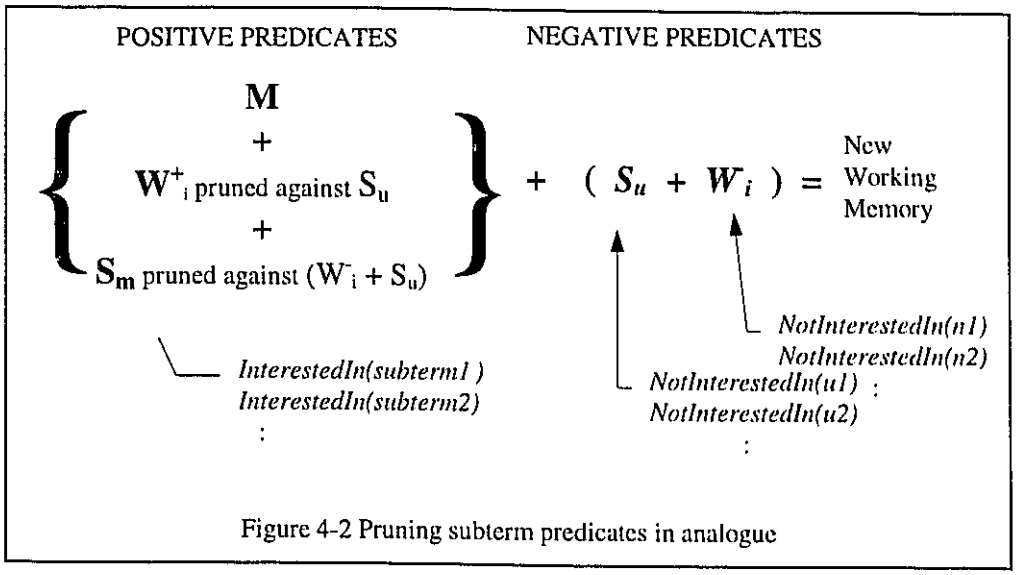

Figure 4-2 Pruning subterm predicates in analogue

$$W^{-}_{i+1} = W^{-}_{i} + S_u$$

where W^{+}_{i+1} is the pruned set of the working memory.

Figure 4-2 illustrates the pruning process. The resulting working memory contains both negative and positive predicates, the latter being pruned as shown. For example, in Figure 4-1, the following positive predicates are generated:

InterestedIn(method,"removeIndex:")

InterestedIn(method,"removeLast")

InterestedIn(subterm,"remove")

InterestedIn(subterm,"Last")

InterestedIn(subterm,"Index")

where S_m contains the last three *InterestedIn(subterm)*. Negative features are also generated for each unmarked method. In working memory, the following NotInterestedIn (subterms = S_u) are added and used later for pruning positive predicates:

NotInterestedIn(subterm,"add")

NotInterestedIn(subterm,"after")

NotInterestedIn(subterm,"Index")

NotInterestedIn(subterm,"Last")

When pruning occurs, the following two InterestedIn() predicates are removed:

InterestedIn(subterm,"Last")

InterestedIn(subterm,"Index")

Since the only purpose of NotInterestedIn predicates is to prune positive features, they do not carry any weight unlike the InterestedIn predicates. Pruning is processed by the following rule:

If	(NotInterestedIn(subterm <subterm_name>) and
	InterestedIn(subterm , <subterm_name> , <cf>))
Then	(remove InterestedIn(subterm , <subterm_name> , <cf>))

Figure 4-3 Pruning subterms rule of inference

In effect, every *InterestedIn(subterm)* predicate that has a matching *NotInterestedIn(subterm)* predicate (i.e. based on same subterm name), is pruned from the augmented working memory.

The template, which is the form of the analogue used for matching, contains only positive features, whereas the analogue includes both positive and negative features. The algorithm for matching library items against the template takes into account the subterm predicates. As well as scoring class items on method names and class names, each library item gets a score for implementing a method whose subterms are in the analogue as an InterestedIn predicate.

Scoring of class items on subterm is done as follows. For each class node in the library, the method names are split into subterms and added to the class node as a list of subterms. For each subterm template, a class item gets a score of 1 if the subterm is present in the class subterm list, and 0 otherwise. This method has the advantage of being fast as the library is preprocessed and each class node subterm list is generated when the node is created. The total score of a class item is then

(score on method names) + (score on subterms) + (score on class names).

4.4 Experimental results

The effectiveness of negative inference is demonstrated experimentally. The general method is to compare the performance of the original active browser (Base) to an active browser with negative inference (Neg). Active browsing performance is measured in two ways:

1. by the wins/losses statistic which counts the number of times the active browser correctly infers the target before/after the automated agent (Rover, see Chapter 3) finds the target.
2. by comparing the search lengths of Base to Neg, i.e. the number of steps before Neg and Base infer the target class.

The Smalltalk code library was used as the test library. Each of the 389 classes, was set as the search target. The implemented methods and class name of the target class define the search specifications of the Rover. The Rover search is started and the ranking of the target in the suggestion box (the active browser's guess) is compared to its ranking in the browser (Rover search) at every step. Two sets of experiments were conducted:

1. Rover search with Base active browser
2. Rover search with Neg active browser

Out of the 389 classes, there are a number of invalid runs. A run is considered invalid when the search is either too long or too short, where the length of a search is the number of steps before search stops. A run is too long when neither Rover nor the active browser (Base or Neg) have found the target after a certain number of steps, which is set to 70. The motivation here is to avoid infinite searches, as for example when the target is a unique class which has little in common (methods or class name) with the rest of the library. A run is too short when the search of Rover ends in fewer than 5 steps, the number of steps for the target to be in the suggestion box before the active browser can win. In Base, there are 69 short and 17 long runs. In Neg, there are 69 short and 10 long runs. The total number of valid runs is 299 Base and 306 in Neg. The difference is accounted for by the smaller number of long runs in Neg, indicating that with negative inference, active browser reduces search length.

	valid runs	invalid runs	
		short	long
Base	299	69	17
Neg	306	69	10

Table 4-1 of number of valid/invalid runs

4.4.1 Comparing win/loss/draw of Base and Neg

Wins, losses and draws count how many times the active browser finds the target before, after or at same time as Rover respectively. Refer to chapter 3 for a definition of each of these terms. Table 4-2 displays the counts for both Base and Neg.

Table 4-3 expresses the wins, losses and draws, as a percentage of total number of valid runs. The percentage win figure is similar to the probability of a valid search ending in a win.

The Base active browser wins in 70 searches while Neg wins in 161. The number of wins is more than doubled. Similarly, the loss count improved in Neg. The number of losses in Base (207) is almost twice that in Neg (110). The better win percentage indicates that during any search, the probability that Neg finds the target before Rover is twice that of Base finding it before Rover.

	Wins over Rover	Losses to Rover	Draws with Rover	Total valid runs
Base	70	207	22	299
Neg	161	110	35	306

Table 4-2 Neg v/s Base

	Wins over Rover	Losses to Rover	Draws with Rover
Base	23.4 %	69.2 %	7.4 %
Neg	52.6 %	36.0 %	11.4 %

Table 4-3 percentage wins loss draw

4.4.2 Comparing Base and Neg search lengths

The search length for a target is the number of steps before the active browser (Neg or Base) infers the target. By comparing the search lengths of Base and Neg, we are directly comparing which agent is better, regardless of whether their respective search ends in a win or loss against Rover. On the other hand, wins versus Rover is not a direct comparison of the performance of the active browsers.

Table 4-4 shows comparative search length statistics for Base and Neg. Each row corresponds to the runs in which one system finds the target faster than the other. The first column of the table shows the number of runs in which the corresponding active browser

is faster than the other. Out of the 389 runs, Base is faster than Neg in only 11 cases, whereas Neg searches ended before Base 120 times. In the other 258 runs, both active browsers' searches are of the same length. This result has greater significance than the win stats. In 91.6 % of the 131 runs (where one's search length is better than the other), Negative inference reduced the search length.

	Number of runs	Average search length		
		Base	Neg	diff
Base faster	11	9.9	11.5	1.6
Neg faster	120	23.6	12.6	11

Table 4-4 Comparing Base and Neg average search lengths

In the 11 runs where Base is faster than Neg, the average length of searches in Base is 9.9 steps and 11.5 steps in Neg. In these runs, although Neg is slower than Base, it is lagging behind by only 1.6 steps on average.

In the 120 runs where Neg is faster than Base, the average length of searches in Base is 23.6 steps and 12.6 steps in Neg. On average, Base would require 11 more steps to find the target than Neg. In effect, Negative inference reduces the average search length by 46.6%.

These results show that negative inference and the use of subterms is rarely detrimental and never a significant impediment. Most frequently, it almost doubles the speed with which the active browser finds the target. In the few cases where Neg actually fares worse than the Base system, the difference in inference between the two is negligible.

4.4.3 Comparing Base to Neg on active searches and target average rank

Similar observations can be made from the following graphs. They represent graphically the difference between the active browsers. Figure 4-4 plots the number of active searches for Rover, Base and Neg at each step. A search is active at step n if Rover or active browser has not found the target after n steps.

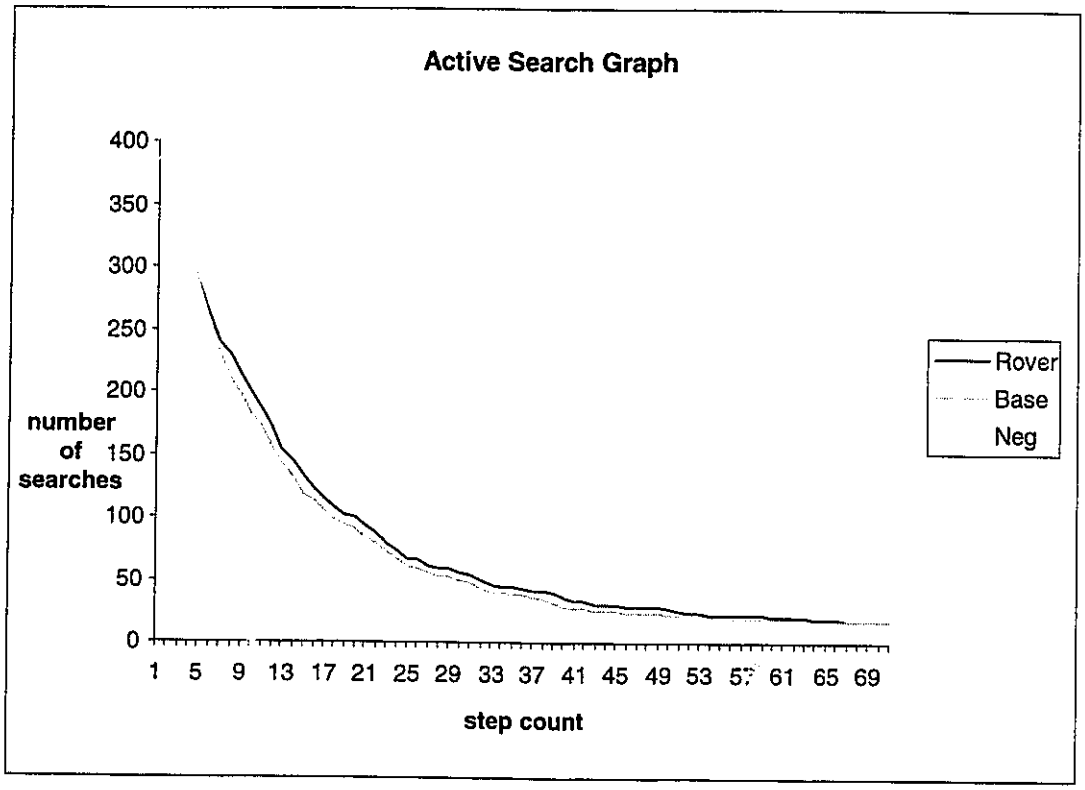

Figure 4-4 Active Search graph

After only a few steps, the agents find many targets. After 10 steps, about half of the searches are completed by Rover. The curve for the active browsers are necessarily lower than that of Rover because search ends when either Rover finds the target or they find the target before Rover. Hence, the vertical distance between Rover's curve and any active browser's curve, at step N, represents the number of targets that the active browser has identified that Rover has not found yet. Note that invalid runs are included in the number of active search as well. For the first five steps, the curves are all identical simply because by definition, the active browsers can only win after 5 steps. At the other extreme, all the curves decrease to an asymptote equal to the number of "long" or "invalid" searches for each agent (17 for Rover and Base, 10 for Neg).

The curve for Base is close and parallel to Rover. This indicates that Base and Rover are completing searches at similar rates. On the other hand, Neg completes searches at a much faster rate from steps 5 to 20. The vertical difference between Neg's and Rover's curve is constantly increasing from step 5 to step 20. This observation reinforces

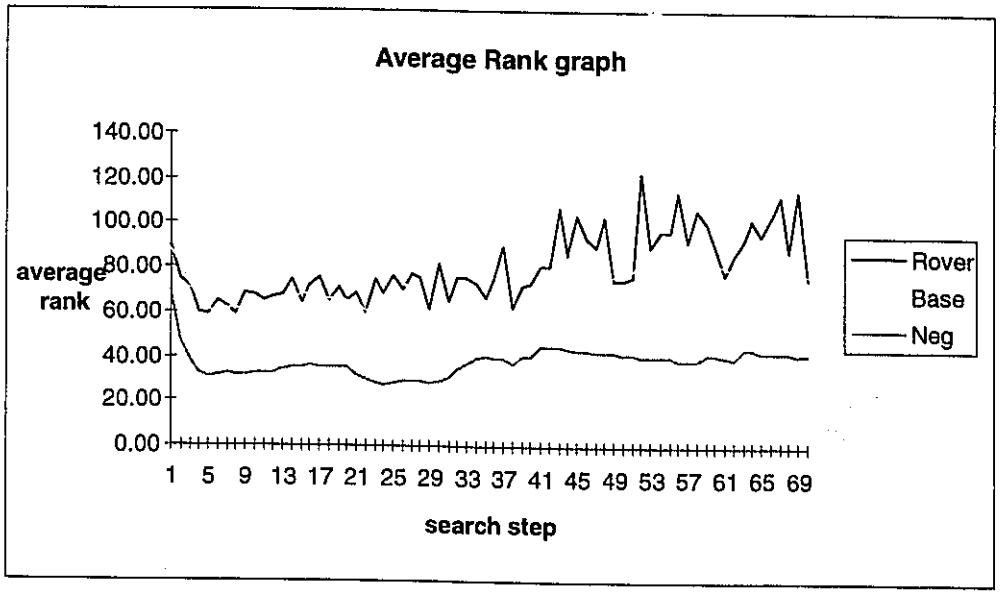

Figure 4-5 Average rankings of Base, Neg and Rover

a fact that is apparent in Table 4-4: namely, that Neg's improved performance over Base manifests itself primarily in the first 20 steps.

The preceding graph showed how fast an agent reaches the target, but does not show how the rank of the target evolve during search.

Figure 4-5 plots the average rank of the target at each step. The average for step N is the ratio of the sum of the rank of active targets at step N, to the number of targets still active at step N. The best rank is 1; the higher the average rank, the worse the system. Base is much better than Rover. Except for the first 5-10 steps, the target is about 5-15 positions better in Base than in Rover.

From this perspective, Neg dramatically improves the quality of Base's inference. After the first step, Neg ranks the target 20 positions higher than Base, and this difference increases to 30 after the first few steps. At every step, the average rank of the target in Neg is almost half of the average rank of the target in Base.

The two graphs presented in this section gave two different perspectives of the experimental data collected for the active browsers and Rover. Both graphs, in different ways, indicate overwhelmingly the improvement of negative inference and subterms on the original active browser. The active search graph showed that Neg was completing

searches faster than Base. This result is reinforced by the fact that Neg ranking of the target is half that of Base, as can be seen in the average rank graph.

4.5 Discussion and conclusion

Our method of negative inference preserves the non-intrusive characteristic of active-browsing. Contrary to most negative inference systems (see Chapter 7), where the user is part of a feedback loop and is required to provide direct feedback to the inference system, negative features are inferred as a result of the user's actions while browsing. It does not disrupt the user's interaction with the browser which could otherwise hinder the user's search process and reduce the effectiveness of browsing. Moreover, from the user's standpoint, the interface and interaction with the browser is identical to the Base system while active browsing performance is increased.

Experiments were conducted to evaluate this implementation of negative inference using an automated search agent (Rover). Results show that the performance of active browsing is more than doubled when using negative inference. With negative inference, the active browser finds the target, before Rover, in 52% of its searches while without negative inference, the active browser succeeds in only 24% of all targets. On targets where Base and Neg performance differs, negative inference reduces search length by eleven steps in 91.6% of the searches. In short, the active browser with negative inference and subterms finds the target before Rover twice as often as without, while reducing the speed to find the target by almost half. Plots of the active browsers performance reinforce these facts, and indicate that the improvement occurs mostly in the first 20 steps of a search, and on average, the position of the target in the suggestion box is reduced by half.

Chapter 5

5. Selective Search

In previous chapters, we have shown how the accuracy/performance of the basic active browsing paradigm can be improved by inferring negative features. Another area that can be improved is the speed of evaluating every item in the library in order to calculate the new suggestion box, a cpu intensive process which can slow down the system noticeably. In this chapter we propose a method called "selective search" that improves the response time of the active browser. Section 5.1 describes the general problem and motivation. In section 5.2, we provide an analysis of the dynamic behavior of the library during an active browsing session. These insights will best explain the idea behind the selective search method, which is described in section 5.3. Section 5.4 presents an implementation of that method. The last two sections, provide an analysis of the algorithm and areas of improvements. Experimental evaluation is presented in chapter 6.

5.1 Problem Description

Whenever the user performs an action in the browser, the active browser updates the suggestion box, i.e. its guess of the ten classes that best match the user's search target. This process involves searching the whole library for the highest scoring items by evaluating each one against the system's inferred template. In large libraries, the matching of the template to library items can be very time consuming and is effectively a bottleneck in the active browsing process.

The active browser is by design non-intrusive. The inference component does not require any explicit feedback or input from the user. Nonetheless, such a system can be intrusive if it disrupts the user's normal search process. In some cases users may wait on the system's suggestions before selecting the next browsing action. In this case, any significant delay in updating the suggestion box would be disruptive to the user. Maintaining the non-intrusive property of the active browser requires that the suggestion

box is updated without perceivable delay to the user. When the library is large, the search process can be noticeably slow and cause significant delay.

One solution to this problem is to limit the system's search to part of the library at every update of the suggestion box. By evaluating only a fraction of the library, we effectively reduce the number of computations required and the time taken by the active browser to update the suggestion box: the fewer items evaluated, the greater the gain achieved. The resulting ordering of the suggestion box is an approximation of the case when every item is evaluated. Hence, there is a risk that this gain in speed is accompanied by a reduction in the system's accuracy in inferring the user's target. Our goal is to maximize the former while limiting the latter.

5.2 Dynamic behavior of Library rankings

At each iteration, when new template terms are generated from the user's most recent browsing action, every item of the library is scored against these terms and sorted according to their score in increasing order. The active browser then displays the ten highest ranked items in the suggestion box.

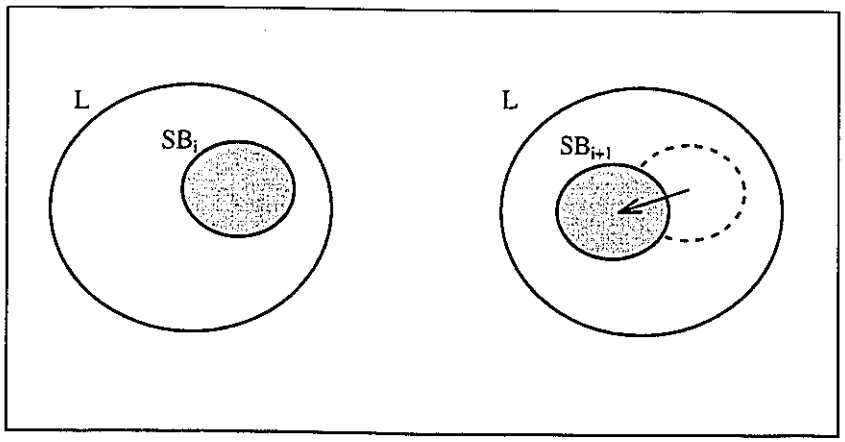

Figure 5-1 Shifting of "best" items after a library update

Figure 5-1 shows the state of the suggestion box items, before and after the *ith* update of the suggestion box (also referred to as SB in this chapter). L is the set of items in the library and SB_i the set of items in the suggestion box at iteration i. At the next

iteration $i+1$, when the user performs an action, the system updates the score of all items in the library: SB_{i+1} is the resulting set of the ten best scored items. As a result of this "search", the shaded subset SB has shifted, i.e. the suggestion box now displays different items, or orders them differently. The overlap between SB_i and SB_{i+1} indicates that some items may stay within the suggestion box after the $(i+1)th$ iteration.

At each iteration, SB can shift in three ways. First, the suggestion box could remain unchanged. This occurs usually when the browsing search is well focused, so that new templates term are consistent with older ones, and updating the scores of all library items neither changes the set of the ten highest scoring items nor the order of the best ten. Second, the new SB can be a different set from the previous one. None of the new highest ten items appeared in the suggestion box in the previous search. This occurs when there is a dramatic shift in the user search, generally at the beginning of a search when the user is unsure. Third, most of the time, only a few items change in the SB.

At every iteration, the template can be updated in two ways. First, new terms are added to the template when the user shows interest in a new term. This can result in significant shifts in SB. Second, the weight of an existing template term is increased when the user shows renewed interest in a feature. The template is not augmented with new features, and consequently, either brings about a small shift of SB or a simple reordering of items in the suggestion box.

On the whole, due to the incremental nature of the template matching process, dramatic shifts of an item ordering (ranking on score) are few and in general SB shifts tend to be localized. In all three types of SB shifts, the ordering of the items in the subset $(L - SB)$ changes too after each iteration. Set $(L - SB)$ contains items in L not in SB, where the minus sign,"-" denotes set difference.

Although elements in $(L - SB)$ are not visible to the user, they are updated so that those ranked close to the SB might show up in the suggestion box in later iterations. As well, an item can improve its ranking dramatically if the new template terms are a good match to that item's features.

One can observe that the most work accomplished during search is updating the "invisible" items, i.e. items not appearing in the suggestion box. The subset of items

ranked lower than 10 is large and slows down the process of updating the SB. However, it is necessary to evaluate these items as they may gradually appear in SB in later iterations.

Limiting search to part of the library implies deciding which items should be evaluated at every iteration. Any solution should take into consideration the two following facts:

- most changes are localized i.e. the next best items are more probably ranked closely to the suggestion box.
- lower ranked items need to be evaluated as well, as they can appear in the SB later.

5.3 Selective Search, an abstract solution

In this section, we describe the method of selective search (search in the context of evaluating nodes in a graph, and identifying the n nodes with the highest score). The general strategy of selective search is to evaluate, at each iteration, only a subset of the graph.

In a graph containing n nodes, only k nodes are evaluated at each iteration, thus achieving a reduction in computation of a factor of roughly n/k. One would normally expect a tradeoff between the gain in search speed and the decrease in the active browser performance, as measured by the active browser's rate of success in guessing the user's search target. In order to achieve this reduction with the least penalty on the active browser's performance, selecting the right items is critical.

5.3.1 Partial scoring

By selectively evaluating part of the library at every iteration, some items will not be evaluated on some template terms. Hence, each item n has a set of template terms T(n) which has not been evaluated yet. The partial score of item n, say s_n consists of two components; g its exact score on template terms for which n has been evaluated, and h, an upper bound estimate of its "score" on the remaining template terms.

$$s_n = h_n + g_n$$

h_n is computed as the sum of the weights of all template terms n has not been evaluated yet. Every item in the library has the associated information: g, its exact score, its h score and the set T of templates corresponding to h.

The size of T (number of terms in T) is a good indication of the uncertainty associated with the score of an item. The larger the number of unevaluated template terms, the less confidence there is in the score of an item. The uncertainty in the score of an item can be reduced by evaluating the item on some or all terms in its associated set T.

5.3.2 Uncertainty Class and ordering partially scored items

In order to generate a list of the ten highest scoring nodes, we need to define an ordering scheme for partially scored nodes.

There are two possible ways of ranking partially scored items:

1. based on partial-score: item i is ranked higher than j if and only if, $s_i > s_j$.
2. based on upper bound estimate h, using exact score g to break ties.

The first option may lead to inconsistency because an item can be ranked higher than another one if it has a higher h-value but lower g-value. In other words, an item with greater uncertainty in its score can be ranked higher than one with lower score but greater confidence in the score. For instance, consider items i and j, with g value g_i and g_j, and h-value of h_i and h_j respectively. Suppose i is ranked lower than j. If they were sorted on partial score,

then $g_i + h_i < g_j + h_j$

But it can be the case that,

$g_i > g_j$ and $h_i << h_j$

implying that i has a higher exact score than j, but with lower uncertainty.

Option two is our choice for ranking partially scored items in the library. Items are sorted first on their h-value and second, items with the same h-value are sorted on their g-value. Thus i is ranked higher than j if $|T(i)| < |T(j)|$, i.e. i has been evaluated on more template terms than j. In addition, if i and j have identical uncertainty i.e. $|T(i)| = |T(j)|$, i is ranked higher than j if and only if, $g_i > g_j$. Figure 5-2 summarizes the ranking algorithm.

let i, j partially scored items such that

$s_i = g_i + h_i$ and

$s_j = g_j + h_j$

i is ranked higher than j iff

1. $|T(i)| < |T(j)|$ or
2. $|T(i)| = |T(j)|$ and $g_i > g_j$

Figure 5-2 Ranking items with partial scores

Items of equal uncertainty are partitioned into uncertainty classes. We thus define an "uncertainty class" as a set of items which have not yet been evaluated on the same number of template terms. Items in an uncertainty class have identical T and h value. Each class then has an associated T and h value. According to this ranking scheme, uncertainty classes are thus sorted by their uncertainty value (size of T). A higher ranked class is one with lower uncertainty factor (the least h or least number of template terms in T). Also, within an uncertainty class, every item is ranked by their exact score, i.e. g-value. Note that an item can be ranked low (high h) but have a high exact score. These items are good candidates for future selective evaluations.

In summary, the ordering scheme for ranking partially scored items effectively generates a partitioning of the library into uncertainty sets. The top ten items in the highest ranking (least uncertainty) class are displayed in the suggestion box.

5.3.3 Class migration

When selecting items to be evaluated, we have to consider both items close to SB and lower ranked items. Low ranked items, of high uncertainty (i.e. large h value), could potentially rank higher were the uncertainty to be decreased by evaluating a subset of its template set T. Evaluating an item on some term of its unevaluated template set T would move that item from one uncertainty class to a higher order one. This effect is referred to as "class migration". At each update of the suggestion box items are selected from all classes for evaluation. Suppose for example, that L is partitioned in two uncertainty classes, C_1 and C_2, as in Figure 5-3. Class C_1 is of lower uncertainty than C_2 and is ranked higher. Hence, the ten items displayed in the suggestion box are the ten best items in C_1. The shaded areas of each class represent the items selected for evaluation.

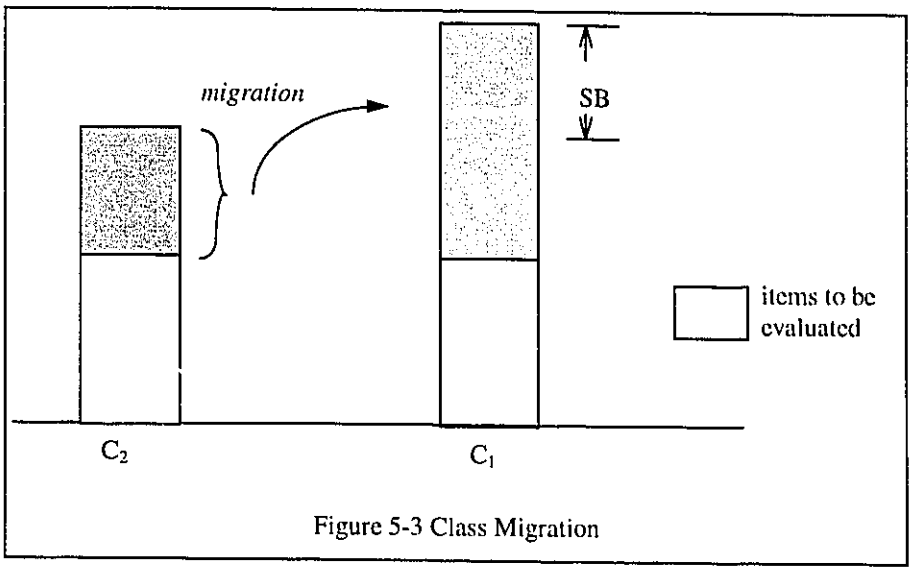

Figure 5-3 Class Migration

Evaluating items in C_1 and C_2 on template terms in each class' respective T improves their uncertainty level and as a result, these items move from a lower order class to a higher order class. As shown in Figure 5-3, the evaluated items in C_2 migrate to C_1. In addition, evaluation of shaded items in C_1 would produce one new class C_0. This process is illustrated in Figure 5-4.

Selective evaluation on one set of template terms is performed as an iterative class migration process. Migration is effected from the class with the highest uncertainty and iteratively to the class with the lowest uncertainty. In Figure 5-4, the process starts with class C_2, from which items are selected and migrated to C_1. At the end of the first iteration, C_2 has reduced in size while C_1 is augmented with the migrating items. In the second iteration, the same process is applied to the augmented class C_1. However, since C_1 is the highest class, a new class C_0 is created and the migrating items are added to C_0. The end result is the partitioning of the library into three new classes. The number of items evaluated is the total number of items that migrated.

This process allows an item from the lowest ordered class to move up by more than one class. An item can thus migrate from C_2 to C_0 in one update of the library. This jump is achieved by successive class migrations. When selective search evaluates k pairs of (node,Template-terms), the same item can be evaluated more than once in one update of the library. Clearly, the main benefit of migration is when such long jumps are realized.

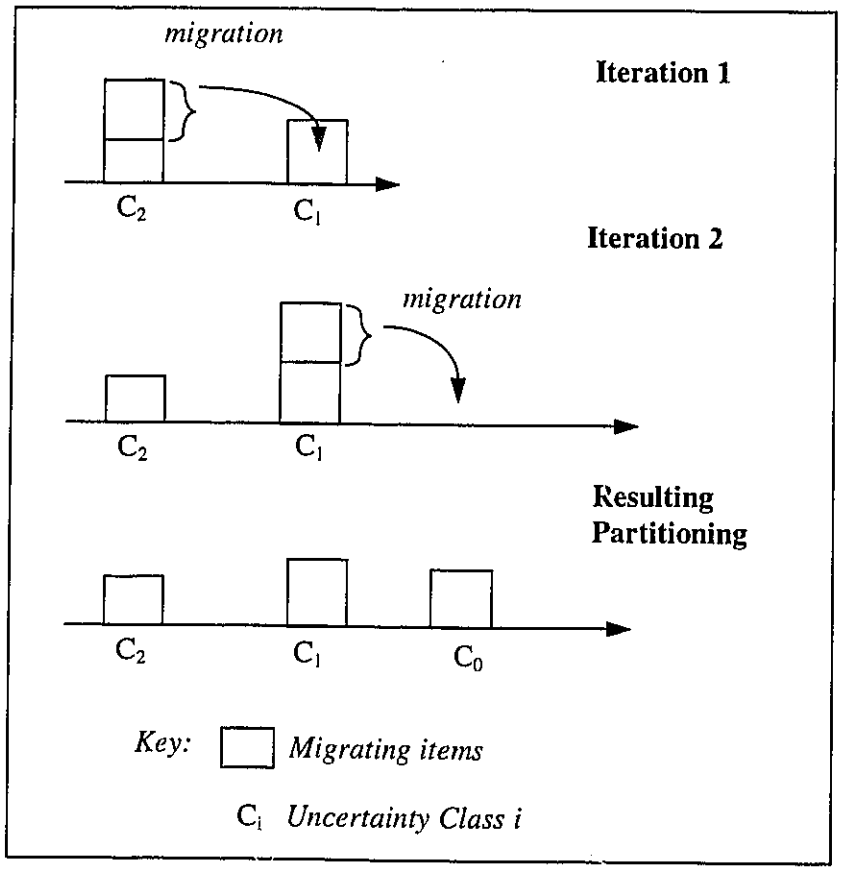

Figure 5-4 Class migrations in one library update

This is analogous to a large shift in the rank of an item when the whole library is evaluated.

The problem with successive class migrations is that at every update a new uncertainty class is created, so the number of classes increases at every update. The goal of selective search is to reduce the amount of work in evaluating library items, but migration introduces the overhead of maintaining the class structure and migrating items from one class to another. The algorithm implemented, as described in the next section, includes a mechanism to limit the number of classes.

5.4 Selective search algorithm implementation

There are two important interdependent aspects to the selective search paradigm: the structuring of the library in partitions and the migration strategy. The partitioning of

the library also requires an ordering scheme of the partitions. The migration strategy, which directly modifies the partitioning at every iteration, describes what and how many items should be upgraded from every class.

5.4.1 Partially Scored Set

Items in the library are partitioned into sets, called Partially Scored Sets (PSS). A PSS is a structure of 3 fields:

- T : is the set of pairs of template terms and their corresponding weight (+ve or -ve) for which items in that PSS have not been evaluated yet.
- H : is the sum of the weights of all template terms in T.
- Item-list : is the ordered list of pairs of library item and its partial score, g-value.

Within each PSS, items are ordered in decreasing value of their corresponding g score. The highest ranked item in a class is the one that has accumulated the highest score on all the evaluated template terms. PSSs are ranked according to their uncertainty level, measured by the size of T. Higher ranked sets have lower uncertainty factor and smaller T. By thus ordering PSSs and items within PSSs, we obtain a linear ranking of the library: first by uncertainty factor and second by g-value. This two tiered ordering simplifies the process of migration.

Figure 5-5 illustrates the partitioning of the library into partially scored sets, the shaded areas representing sorted items within a PSS. In this example, the library items are partitioned into four PSSs. P_i has lower uncertainty than P_{i+1}, thus items in P_i are ranked higher than items in P_{i+1}. Each PSS set P_i, has a corresponding unevaluated template set T_i. Template terms are represented by t_i.

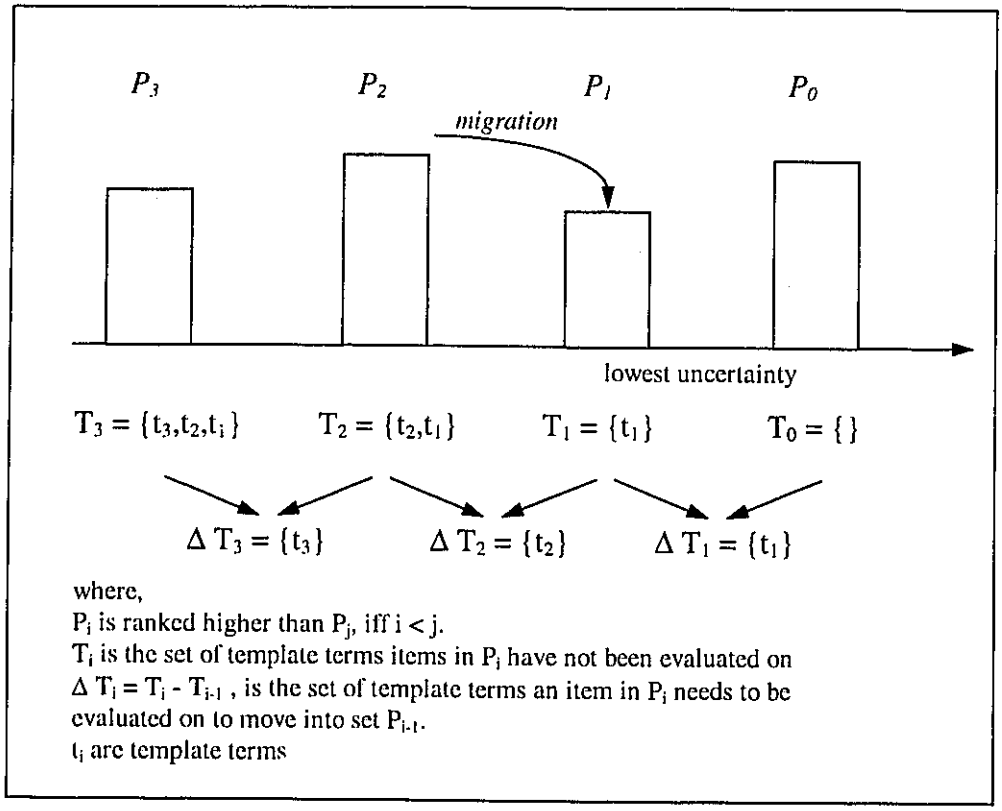

Figure 5-5 Library partitioned into Partially Scored Sets

Let $\Delta T_i = T_i - T_{i-1}$ be the difference, in T, between two consecutive PSSs, P_i and P_{i-1}. ΔT_2 is the set of template terms on which items in P_2 have not been evaluated yet, but on which items in P_1 have. In order for any item in P_2 to move up into the next PSS P_1, it will have to be evaluated on template terms in ΔT_2. Hence, an item from a lower PSS, P_i migrates to a higher one P_{i-1} by being evaluated on the template terms in ΔT_i.

5.4.2 Migration Policy

Let us define the migration level for a PSS as the number of items that migrate from that set to the next higher ranked PSS. The migration policy defines for selective search the migration level for a PSS and the criteria for selecting migrating items from that PSS. In other words, for a given PSS, a migration policy answers the two following questions:

- how many items should migrate from that PSS?

- which items in that PSS should migrate?

The choice of a good migration policy directly affects the performance of selective search. As discussed earlier, low ranked items should be allowed to migrate to upper classes. This can be effected by selecting a particular distribution of migration levels among the classes.

To favor lower ranked items, a "bottom heavy" strategy is needed. In this scheme, migration levels from lower classes are greater than upper classes. On the other extreme, a "top heavy" strategy would favor items in the suggestion box. Both approaches have their pros and cons, and each is appropriate in different situations. In cases where most shifts in SB are localized, the latter is favored, whereas cases in which most shifts in SB are dramatic, the former is most recommended. In this thesis, a uniform distribution is implemented: the migration level for each PSS is of equal value (uniformity is sometimes overridden by the minimum_class_size parameter which is discussed in the next section).

The items migrating from a lower ranked PSS, for example P_2 would end up in P_1 by being evaluated on ΔT_2. The best migrating candidates are clearly the items with the highest g score were they evaluated on ΔT_2. Consider two items a, $b \in P_i$, having partial scores of $s_a = g_a + h_i$ and $s_b = g_b + h_i$ respectively, and scores on ΔT_2 of eval(a, ΔT_2) and eval(b, ΔT_2). Then a is a better candidate for migration than b if:

$$g_a + \text{eval}(a, \Delta T_2) > g_b + \text{eval}(b, \Delta T_2)$$

Since the idea of selective search is to avoid evaluating all items, we do not want to compute eval(a, ΔT_2). Instead, we need an estimate for g_a + eval(a, ΔT_2). The best no cost estimate is obviously g_a. This estimate does not require additional computation. Naturally, better estimates could be obtained by complex methods (e.g. probabilistic or statistical) but at additional costs. Hence, the g value of an item is the chosen criteria for determining migrating candidates in a PSS.

By sorting items in a PSS in increasing order based on their g value, higher ranked items are selected for migration from that PSS. The higher the g value of an item, the higher the probability that that item will rank favorably in the next class, due to its higher partial score.

In summary, our migration policy would specify for a given PSS, say P, in which items are sorted in increasing order of g-value, that the first n items are to be migrated, where,

$$n = \frac{\text{total number of items to be evaluated}}{\text{number of PSSs}}.$$

5.4.3 Selective Search Algorithm

Figure 5-6 describes the selective search algorithm. L is the set of items in the library partitioned into an ordered set of PSS = { P_0 , P_1, .. ,P_n }, where P_i is ranked higher than P_j if $i < j$. P_0 is the initial PSS in which all items are evaluated, i.e. T_0 = { }. When a new set T of template terms is received, L is evaluated selectively. Migrate-list is the set of items migrating to an upper class.

The uniform migration policy is implemented as follows. The parameter k sets a target size for the number of items to evaluate. At each migrating iteration, num_classes is the number of classes left to migrate, and left_to_migrate is the number of items left to evaluate to reach the target k. The target-size, which determines the migrate-list at each iteration, is

$$\text{target\_size} = \frac{\text{left\_to\_migrate}}{\text{num\_classes}}$$

Whenever the top items from the highest ordered class P_0 migrate, a new class is created (in which all items have the template set T = { }, or h() = 0). Thus at every update of the library, the number of classes increases by one. To keep the number of classes under a predetermined limit, small classes are collapsed.

The minimum_class_size parameter is used to determine whether a class is small, and indirectly limits the maximum number of classes. A class is considered too small if either the size of the class is less than the migrating target size, or the size of the class after migrating the target size is less than the minimum_class_size. In both cases, all items in the class are migrated, thus, reducing the number of classes by one. The number of classes is less than $\frac{\text{number of items in library}}{\text{min\_class\_size}}$. The actual maximum number of classes depends on

both k and minimum class size parameter, since k indirectly controls when a PSS collapses (through target size).

```
{ initialization }
left_to_migrate = k
num_classes = sizeof ( PSS )
target-size = left_to_migrate / num_classes
migrate-list = nil

{update h and T of P0 }
T(P0) = ΔT
h = sum of weight of template term in T
{ start migrating iteration }
loop from Pn to P0
        { add migrating items to Pi and sort it }
        Pi = Pi + migrate-list
        sort Pi
        { determine migrate-list }
        if (sizeof (Pi ) < target-size) OR
                {if after migrating target size, remaining class is too small}
           ((sizeof ( Pi ) - target-size) < minimum_class_size)
        then
            migrate-list = Pi
        else
            migrate-list = first ( target-size , Pi )
        endif
        evaluate migrate-list on ΔTi
        Pi = Pi - migrate-list

        { determine next target size }
        num_classes = num_classes -1
        left_to_migrate = left_to_migrate - sizeof ( migrate-list )
        target-size = left_to_migrate / num_classes
        { collapse Pi if empty }
        if empty Pi then
                {add Ti to template list of Pi+1 }
                Ti+1 = Ti + Ti+1
        endif
end loop
{ now update P0 }
T(P0') = { }
h(P0') = { }
migrate top left_to_migrate of P0 to P0'
```

Figure 5-6 Selective search algorithm

5.5 Discussion and analysis

Selective search is a tradeoff between gain in the active browser speed of updating the suggestion box and the accuracy of the system's ranking of the suggestion box items (the more accurate that ranking, the better the inference of the active browser).

The target size parameter k, controls the number of evaluation per iteration. The greater the k value for a given minimum_class_size, the greater the total number of items evaluated at each update, and as one would expect, the more accurate are the scores in the library and the better the active browser.

However, the actual number of items evaluated depends on both k and the minimum class size parameter (mcs). A preset value of mcs may result in the number of items evaluated greater than the target k, when a class is too small (< mcs) and all items in that class are migrated. Thus for the same k value, a greater mcs should increase active browser performance. Different settings of (k,mcs) were evaluated empirically (see next chapter). For example, a setting of (k,mcs) = (20,8) produces approximately the same number of evaluations as a setting of (10,32). The first setting should produce better results granted its higher k-value, but the mcs-value in the second setting is greater than in the first one. The results show that the selective search strategy produces better results in the second setting than in the first. This is strong evidence that k alone does not determine the active browser's performance.

In general one would expect that with selective search the user's search target should rank lower in the suggestion box. However, in some cases selective search may rank the target better than the normal active browser. This can be attributed to the fact that selective migration excludes "bad" candidates (those scored higher than the target in the normal ac' ve browser). Items that would score higher because of some template terms, are ranked low in a class when they are not evaluated yet on these template terms. This is illustrated in Figure 5-7. With selective search, P_0 and P_1 on right hand side, the target is ranked higher than if all items were evaluated as in P. Set *A* are items in P_1 that were not selected to be evaluated and thus do not rank higher than the target in the selective search ranking.

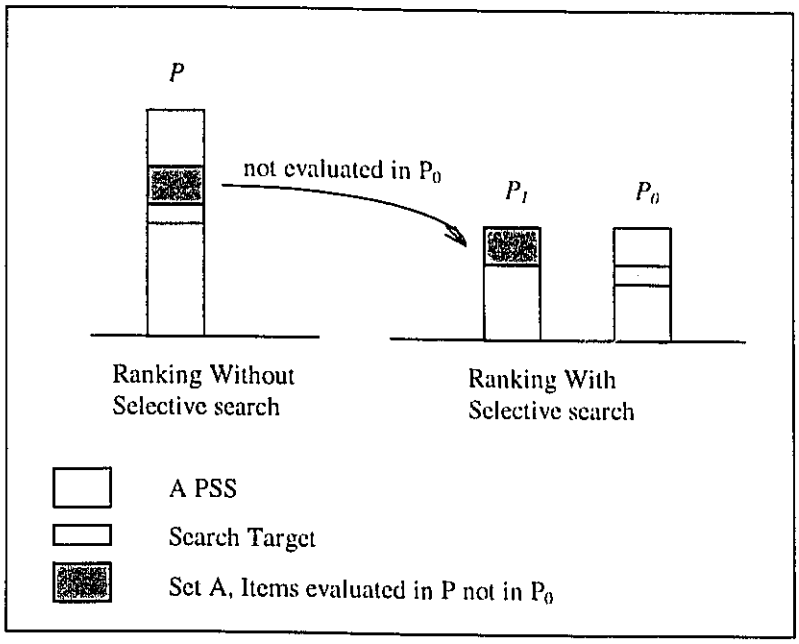

Figure 5-7 Target ranking better with selective search

5.6 Conclusion

We have introduced a new method, selective search, in which the active browser's search through the library is limited to a selected subset of the library. As well, we presented an algorithm for which an implementation has been tested and evaluated. The experimental results are presented in the next chapter.

In this implementation a uniformly distributed migration policy was used. There is scope for further investigation in the variation of the active browser's performance with several factors such as, the migration policy, and class ordering. For example migration levels for each class can be determined based on its h value. Also, classes were ordered per the uncertainty level (size of T) whereas weights of the terms in T may be more relevant.

Chapter 6

6. Empirical Evaluation of Selective Search

To validate the method of Selective Search, experiments similar to the evaluation of the Negative Inference method were conducted. To better gain an understanding of the Selective Search method, active browsers were configured with various degrees of selectivity and the inference performance of each version was measured. Thus, we can determine the level of selectivity which is the most effective.

6.1 Experimental Goals

The goals of this empirical evaluation were to evaluate the effectiveness of the Selective Search method by means of comparison, and to measure how much selectivity affects inference quality. Evaluation of the Selective Search effectiveness was performed in a controlled study to:

- Validate the Selective Search method: Whether Selective Search degrades the active browser's inference performance so much as to be ineffective at inferring the user search target.
- Measure how inference quality varies with selectivity: This would provide the user with a guide of how to tune the active browser for optimal performance.

6.2 Experiment Overview

The experiments were conducted in a similar experimental setting as in the evaluation of Negative Inference (Section 4.4). The Smalltalk library was again used as the application object oriented software repository. The Rover (Section 3.2), an automated search agent, was set up to "browse" through the library for a predetermined target, a class in the Smalltalk library. The active browsers evaluated are versions of Negative Active Browser that include selective search and are generally referred to as "Selective Active Browser" throughout this chapter.

The inference performance of a version of the Selective Active Browser was measured for the whole library, by setting each of the 389 class in the library as the search target. The selective active browser's inference performance on the whole library was measured in terms of the win-loss-draw statistic (Section 3.4.1). The Selective active browser performance was then compared to the Base and the Negative active browser performance. Also, the rankings of the target by Rover and in the suggestion box were collected and used for comparing the different active browsers.

6.2.1 Experiments to measure the effects of the variation of selectivity

To study the effect of selectivity on inference performance, the above experiment was repeated several times with different degrees of selectivity, i.e. percentage of the library selected for evaluation at each update of the suggestion box. This is controlled by setting two parameters: the target migration size parameter k, and the minimum class size parameter *mcs*. The higher the k parameter, the greater the number of items evaluated. Similarly, the higher the *mcs* value for a given k, the higher the number of evaluations. Different versions of the Selective active browser were tested by setting different values for k and *mcs*. Each version is labeled as follows: "*sel-k-mcs*".

Selective active browser	actual average migration per library update	percentage of library evaluated per library update
sel-20-1	20.0	5.1 %
sel-20-5	20.96	5.4 %
sel-10-32	22.37	5.8 %
sel-20-10	23.08	5.9 %
sel-20-40	34.94	9.0 %
sel-40-20	44.96	11.6 %
sel-70-20	73.66	18.9 %
sel-100-40	109.11	28.0 %

Table 6-1 Actual Average migration

Table 6-1 shows the actual average actual migration per iteration for each selective active browser. The selective active browsers are ordered in increasing order of average

amount of migration. Notice that sel-10-32 results in more migration/items evaluated than both sel-20-1 and sel-20-5.

To help understand the effect of k and *mcs*, two sets of experiments were conducted.

The purpose of the first series of experiments, was to study the effect of the k parameter on inference performance. Because a partially-scored-set (Section 5.4.1) of size less than *mcs* migrates completely on an iteration, the actual number migrated can be greater than k. The higher the *mcs* parameter, the higher the difference between the actual number of items evaluated and k. However, this effect is negligible when $k >> mcs$, and the actual number migrated is approximately k. Various degrees of selectivity were set by selecting different k values, and corresponding *mcs* values less than k, $mcs < k$, so chosen as to minimize the effect of *mcs* on the actual number of items evaluated at each update.

In the second series of experiments, our aim was to demonstrate the effect of *mcs* on inference performance. Different versions of the selective active browser were evaluated by varying the value of the *mcs* parameter while keeping a fixed k value (= 20). Additionally, to show the dramatic effect *mcs* can have on actual number of evaluations, a selective active browser with $k < mcs$, sel-10-32, was also evaluated. This version would provide useful comparison when compared to another one with higher k-value but with much lower mcs value, i.e. sel-20-1.

6.3 Experimental Results

Figure 6-1 is a summary of wins and losses for the various versions evaluated. For comparison purposes, the data for the basic active browser, Base ,and the active browser with negative inference, Neg, are also included. The different Selective Active Browsers are displayed on the x-axis in ascending order of average number of items migrated per library update, while the wins, losses and draws are displayed on the y-axis. Base position on the x-axis relative to other selective active browsers is based on win value; Base is located to the right of all active browsers with less wins and to the left of all active browsers with more wins than Base. The win total for a selective active browser represents the number of times the active browser inferred the search target before Rover.

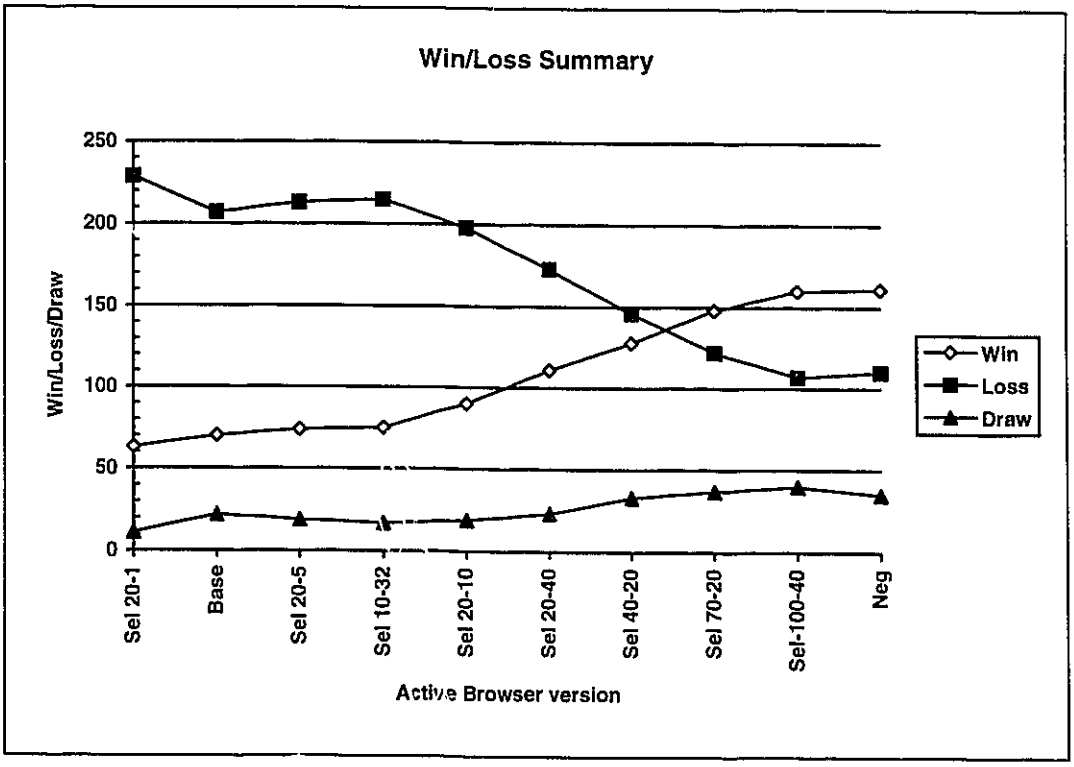

Figure 6-1 Summary Win-Loss-Draw results

The higher the number of wins, the better the active browser at inferring Rover's search target. Conversely, the lower the number of losses is, the better the active browser.

6.3.1 Effect of variation of parameter *k*

Looking at the win curve, from left to right, the number of wins is monotonically increasing. Ignoring Base for now, as the percentage of the library selected for evaluation increases from *sel-20-1* to *sel-100-40*, the number of wins increases. Beyond $k = 100$ no further increase in wins is observed. This clearly shows that selectively evaluating more than twenty-eight percent of the library does not improve the inference quality of active browsing.

This observation is significant proof of the effectiveness of the selective search method. By matching only twenty-eight percent of the library to the template, the active browser can infer the search target as often as Neg, in which all items in the library are evaluated. Effectively, at this point (k=100), the strategy for selecting which items to

evaluate is optimal. The selected items are actually the items that would appear in the suggestion box.

This also shows that the basic active browsing paradigm is highly inefficient. Seventy-two percent of the work in searching the library is wasted. The first twenty-eight percent effort produced the maximum active browser inference. The extra seventy-two percent does not bring any additional improvement in the inference of the active browser. In fact it turns draws into losses. One explanation for this behavior is that selective evaluation may exclude items that match "generic" features inferred into the analogue. When the analogue contains commonly occurring terms, the target may be ranked low because many items, not relevant to the search target, would match the analogue. In the Selective browser, these "noisy" items may not have been selected for evaluation and thus rank lower than in Neg. Selective search helped rank the target higher (see Section 5.5).

The curve (Figure 6-1) shows that inference performance increases with k, when k is less than 100 and *mcs* is less than k. In the experiments where these two conditions are met, as k increases the percentage of the library being evaluated increases as well (Table 6-1). The more items are evaluated, the better the "estimated" ranking of the items in the suggestion box by the active browser and the closer the inference performance is to Neg.

6.3.2 Comparison based on average ranking

The wins count provides a simple metric for comparing active browser inference performance against Rover. But when comparing active browsers to each other, other metrics are useful. According to the definition of a win, the target must be ranked tenth or higher by the active browser for five consecutive steps. However the win count is no indication of how high the target is ranked in the suggestion box. Moreover when the target ranking is greater than ten the win cannot indicate how close or far is the target to the suggestion box compared to another active browser.

Figure 6-2 plots the average rank of the target at each step. The average for step N is the ratio of the sum of the rank of active targets at step N to the number of targets still active at N. The best rank is 1; the higher the average rank, the worse the system. Five different selective active browsers are displayed. The plots for Rover and Neg are added

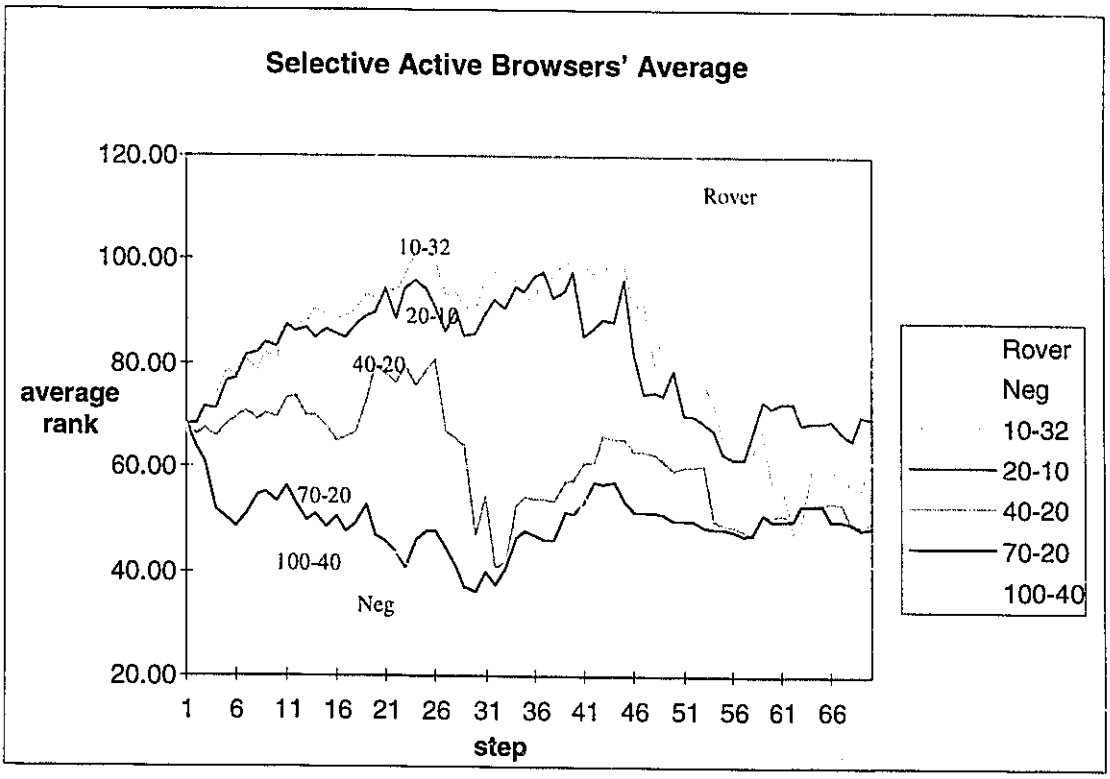

Figure 6-2 Selective active browser average rank graph

for comparison purposes. Consistent with the win graphs (Figure 6-1), the plots show that less selectivity (i.e. evaluating more items in the library) improves the ranking of the target by the selective active browser. This translates into more wins.

Every selective active browser plot is above Neg; the ranking of the target in the suggestion box is on average worse in selective browsers than in Neg. Selective search evaluates only part of the library, thus the ranking of items in the suggestion box is at best an approximation of Neg's suggestion box where all items are evaluated. Even sel-100-40, which has similar number of wins to Neg, does not, on average, rank the target better than Neg at any step of the search. Although selective search can produce good win performance compared to Neg, it does so at the expense of a poorer ranking of the target.

Notice that three Selective active browsers' plots are higher than Rover; *sel-20-10* and *sel-10-32* before step 43 and *sel-40-20* before step 26. The ranking of the target by these Selective active browsers is on average worse than Rover, in the first forty-three steps of the search for the first two, and for the first twenty-six steps for the latter. For searches during these early steps, such Selective active browsers are useless as Rover

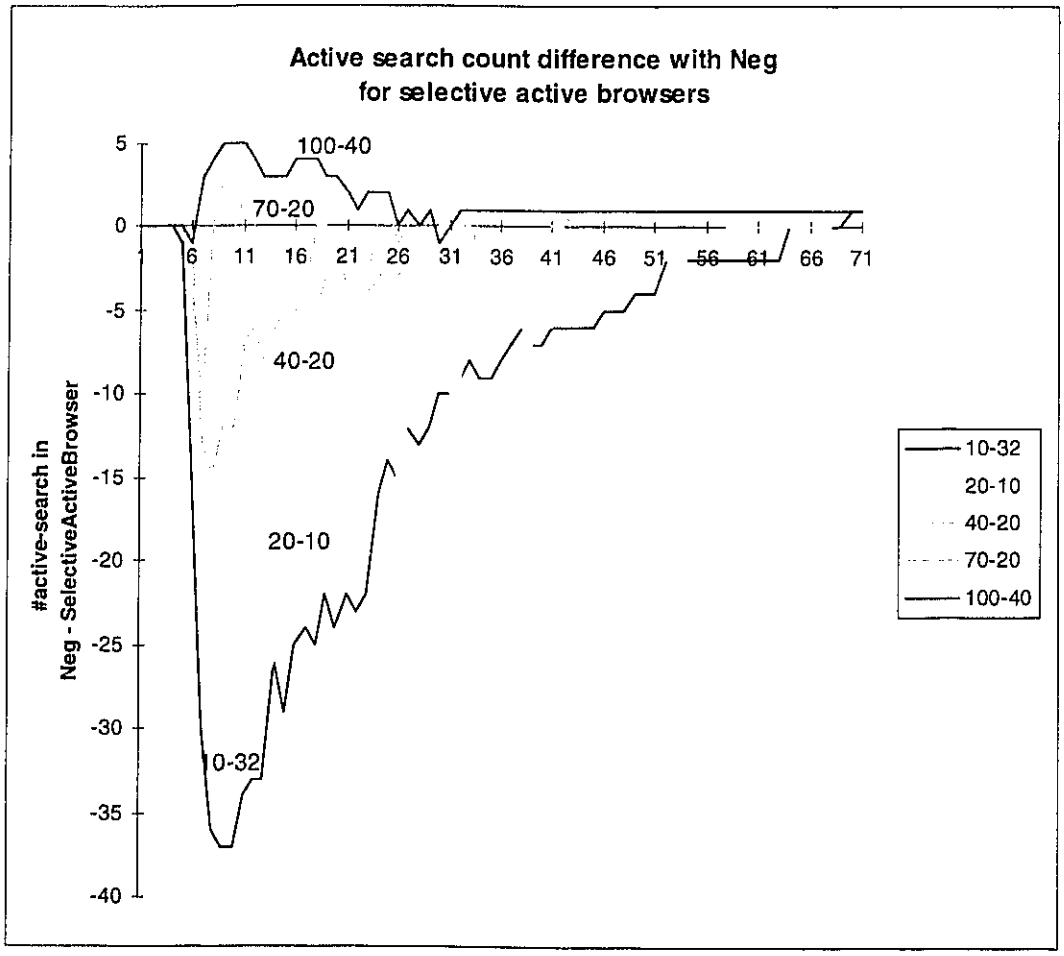

Figure 6-3 Active number of searches

would most likely find the target before the active browser infers it. On the other hand, plots for the other active browser are lower than that of Rover: they consistently rank the target higher than Rover. We can infer from these observations that some minimal percentage of the library needs to be selectively evaluated for the active browsers to be effective; at least more than 11.6% which is the actual percentage evaluated in *sel-40-20.*

The average rank graph revealed the following:

- ranking of the target improves when more items are evaluated
- Neg ranks the target consistently higher than any selective active browser tested
- A lower bound for selectivity is around 11.6% (*sel-40-20*) as selective browsers with lower selectivity are outperformed by Rover.

Figure 6-3 displays a variation on the active search graph (Section 4.4.3). For the purpose of clarity, we plot the difference in the number of active searches between Neg and each Selective active browser. Because fewer number of active searches indicates a better active browser, a higher positive value is better for selective search; the plots are calculated relative to Neg. A positive value at any step, means more searches are active in Neg at that step than in the Selective active browser.

Again here, the ordering of the plots is consistent with that of all the graphs. The higher the number of items evaluated, the higher the curve and the better the Selective active browser. Most graphs are negative , indicating they do not infer the target as often as Neg.

On the other hand, *sel-100-40* is positive between the sixth and the thirtieth step. This means that *sel-100-40* completes more searches than Neg, even though on average, only a 28% of the library is evaluated at each iteration. This accounts for the higher number of wins for *sel-100-40* than Neg as displayed in Figure 6-1. This supports the hypothesis that some non-win searches in Neg can be converted into win with the selective search method (Section 5.5).

6.3.3 Effect of the *mcs* parameter

The second set of experiments were conducted to study the effect of the mcs parameter on the inference performance of a selective active browser. Different versions of the selective active browser were evaluated with a similar k value (20) but different *mcs* values. As discussed earlier (Section 6.2.1), the effect of *mcs* is to increase the actual number of items evaluated at an update of the suggestion box to a higher number than the target k. From Table 6-1 it can be seen that for k=20 and increasing *mcs* from 1 to 40, the average number of evaluations increases from 20.0 to 34.94.

Considering only *sel-20-x* plots in Figure 6-1, it can be seen that for the same k value, by increasing the value of the *mcs* parameter, the number of wins increases. Figure 6-4 shows the effect of *mcs* on the ability of the selective active browser at finding targets.

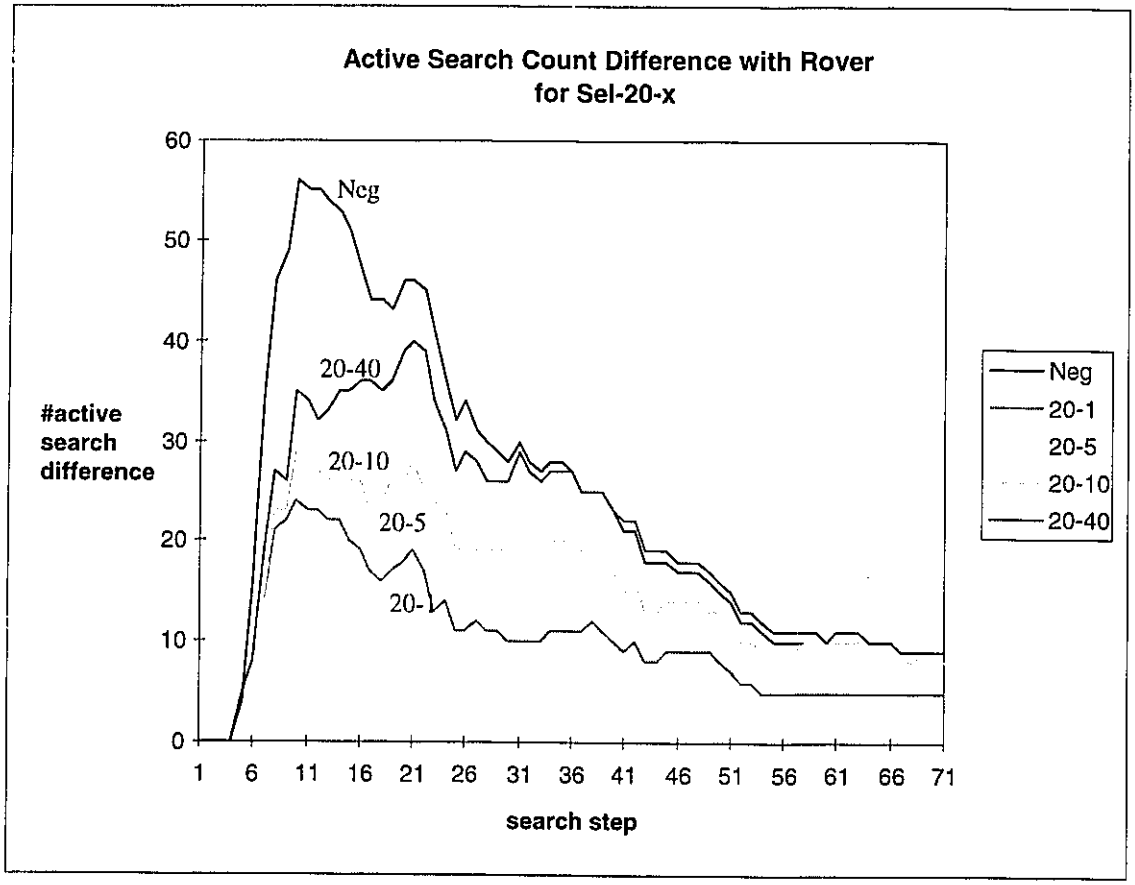

Figure 6-4 Variation of *mcs* parameter

Figure 6-4 plots the difference between the number of active Rover searches and the number of active searches in the Selective active browser at a given step. The y-value for an active browser represents for a given step, the number of targets which Rover has not found yet, but which the active browser has already inferred: the higher the y-value, the smaller the number of active searches in the active browser, and the better the active browser.

From the graph, it can be seen that the higher the *mcs* value, the higher is the curve. The vertical gap between two plots represents the number of searches that the higher selective active browser has completed, but which the lower one has not. In other words, the higher the value of the mcs, the better is the selective active browser.

6.4 Conclusions

The empirical results clearly demonstrate the effectiveness of the selective search method in reducing the total number of evaluations for matching, with little penalty to inference quality. The results have shown that by performing only 28% of all the work during evaluation of the library items, the active browser can achieve similar results as when evaluating the whole library. This implies that the Base active browser is highly inefficient, and selective search at best can reduce the amount of work performed by an active browser by a factor of four, in order to achieve the same inference performance. However, it was found that the target ranking with selective search is on average poorer than exhaustive evaluation.

The results and graphs clearly show the effect of *mcs* and *k* on the inference performance of the selective active browser. In essence, the higher *mcs* or *k*, implies that more items are evaluated (i.e. higher degree of selectivity) and thus greater probability for the selective active browser to infer the user's search target. However when both parameter vary the resulting effect may be unexpected. For instance, *sel-10-32* evaluates on average 5.8% of the library while *sel-20-1* evaluates 5.1%. Consequently the former wins more often than the latter although having a smaller *k*. The effect of the *mcs* parameter on inference performance is greater in *sel-10-32* than in *sel-20-1*.

Selective search is effective for k values between 40 and 100, i.e. when roughly between 12% and 28% of the library is evaluated. Between these two bounds, a selective active browser can be configured or tuned to the user's preference and depending on the size of the library. With higher selectivity better inference can be achieved.

These experiments have demonstrated the effectiveness of the selective search method, but at the same time, allowed us to understand better the intricacies of the selective search algorithm and the way it can be tuned to optimized performance. Clearly, the conclusions presented here are not definitive but suggest ideas for future research for which more extensive experiments can be devised to study various aspects of the selective search methods.

Chapter 7

7. Related Work

The rationale for our research is to promote software reuse by providing an effective software retrieval system. The problem of retrieval is discussed in section 7.1 and similar systems presented. The active browser can be classified as a personal assistant as it helps the user in the browsing activity. Section 7.2 introduces personal assistants and discusses some examples. The third and last section, emphasizes the inference aspect of active browsing, and presents Programming By Demonstrations (PBD) systems and plan recognition systems, where goal inference is central.

Active browsing is a software retrieval system as it assists users in the task of retrieving software components. Similar software retrieval systems (section 7.1.2 and 7.1.3) focused on classifying software components using indexing methods or knowledge-base methods. On the other hand active browsing relies on the intrinsic relationships of object oriented code and does not attempt to classify software artifacts in the library.

Our system monitors user actions during a browsing session and infers the user's search target. PBD systems, personal assistants and critiquing systems are similar systems that monitors user actions, and present suggestions to the user. However these systems differ on the format of the suggestion they provide and the aspect of the user-system interaction for which assistance is provided. Plan recognition systems, likewise, infer the user's intent from user actions but the result of the inference is the user's plan.

7.1 Reuse systems

Although there is a great diversity in the software engineering technologies that involve some form of reuse, there is a commonality among the techniques used. Four dimensions are mentioned in [Biggerstaff and Richter, 87]: abstracting, selecting, specializing and integrating. The focus of this chapter is on the selection dimension.

7.1.1 The Selection problem in software reuse systems

Among the truisms that [Krueger, 92] found as being difficult to achieve in practice is "To reuse a software artifact effectively, you must be able to "find it" faster than you could "build it"." (p. 143).

Traditional approaches to software retrieval fall into two complementary categories: high-level classification techniques, which emphasize retrieval by software category , and low-level cross-reference tools, which facilitate various kinds of browsing at the code level. High-level classification can follow basically two approaches. The classification can be extracted from the component itself (code, documentation) using semi-automatic indexing methods. Or, on the other hand, a knowledge based classification scheme can be implemented using information about the components that lies outside of them.

7.1.2 Retrieval by Automatic Indexing methods

This approach extracts information from the natural-language documentation of the software components. The goal of this approach is to characterize each component by a set of indices that are automatically extracted from its natural language description. Several systems have implemented this approach.

[Fraser et al., 89] proposes a system where keywords are automatically extracted from the documentation of every component. They are extracted from the "comments" for each method in the Smalltalk library. Common English terms are removed and remaining words form a set of keywords. Queries can be formed by combining the keywords using logical connectives.

Similarly, [Matwin and Ahmad, 94] constructed a reuse system by extracting indices from program comments of a linear algebra library of modules not designed for reusability. Names of concepts, which are the basis of the application library domain model, are acquired by AZTEC, a natural language parser. REMIND, a Case Based Reasoning (CBR) shell is used to create a case library incorporating this domain knowledge.

[Frakes and Nejmeh, 87] uses an existing Information Retrieval system, CATALOG, for storing and retrieving C software components. Each component is characterized by a set of single-term indices that are automatically extracted from the headers of C programs.

[Maarek et al., 91] automatically extract indices, through a more complex analysis of text, in the form of lexical affinities. In linguistics, a lexical affinity is the relationship between two words, and represents the correlation of their appearance in a phrase. The indexed objects are assembled into a browsing hierarchy using a "hierarchical clustering" technique which draws information exclusively from the lexical affinities. This approach has been implemented in the Guru system to organize an AIX utilities library. In [Helm & Maarek, 91] the same technique has been applied to an object-oriented class library.

The automatic extraction of classification information presents advantages in cost, transportability and scalability, but can't substitute for meaning. Such approaches do not use any semantic knowledge and do not intend to understand the documentation or the source code. However much information can be obtained from browsing the data and control flow graphs of software components. This approach is used extensively in object oriented programming [Deutsch, 89; Liskov, 87]. The code analysis approach is also adopted in CAESAR [Fouqué and Matwin, 92], which uses the CBR approach as in [Matwin and Ahmad, 94]. In CAESAR, cases consist of C source code and program specifications.

7.1.3 Knowledge based approach to software retrieval

A key feature of knowledge based retrieval systems is that it draws semantic information about software components from a human expert. This approach requires domain analysis and a great deal of pre-encoded, manually provided semantic information.

Prieto-Diaz's classification scheme [Prieto-Diaz, 91], is based on library science. He proposes a set of six facets, three related to the functionality of the component and three related to its environment. The different values a facet can have are called terms. Each component is characterized by a six-tuple of terms. To classify a component, a value for each term must be given. Search in the reusable library is accomplished by formulating

a query with six terms. The conceptual graph that organizes domain concepts represents manually encoded knowledge about the domain.

AIRS [Wong, 92] is an example of a semantic-net-based software retrieval system, that supports *incremental search* by using the version space search strategy. Incremental search is a type of query refinement in which search constraints are added incrementally to eliminate search candidates. AIRS uses the concept of subsumption to organize software descriptions into a generalization/subsumption hierarchy. It uses a heuristic retrieval algorithm based on a numerical conceptual distance measure which the user has to specify.

LaSSIE [Devanbu et al., 91] is a frame-based knowledge representation. Software components are described in terms of the operations they perform. Each operation is described by giving its actor, object, recipient, agent, environment, and so on. These relationships are coded in a specialized knowledge representation system which classifies them into a conceptual hierarchy. The four principal objects are OBJECT, ACTION, DOER, and STATE. Nodes below DOER and OBJECT represent the architectural components of the system. Nodes below ACTION represent the system's functional components. The relationship between the two system components is captured by various slot-filler relationships between ACTIONs, OBJECTs and DOERs. This taxonomic hierarchy ensures that components are properly organized and categorized. The taxonomy can also be useful in query formulation and reformulation. When querying the database, if there are no answers or if there are too many answers, the hierarchy can be used to specialize, generalize, or look for alternatives for an appropriate portion of the query; modify this portion and query again. LaSSIE incorporates a natural-language interface which uses a list of compatibility tuples to parse the input. Compatibility tuples, which indicate plausible associations among objects , are obtained from the frame-like knowledge representation mechanism used by LaSSIE.

CODEFINDER [Henninger, 93] uses retrieval by reformulation. The system uses an associative spreading activation through a graph of keywords and library items in which the arcs represent mutual association. Items with the highest activations and keywords activated during the process are both returned. These same keywords can then be used by

the user to modify his query. The incremental form of query modification is similar to the notion of browsing.

While automatic indexing is more suited to design documentation and the knowledge based approach to source code, [Fernandez-Chamizo et al., 93] proposes a hybrid approach to software retrieval to maximize the advantages of both methods. The class library documentation constitutes the document database. Maarek's method is used to characterize each document by detecting lexical affinities. A simplified version of Prieto-Diaz's faceted classification scheme is used to characterize the software components. This classification is constructed by a deep analysis of the source code of the components.

All these knowledge based systems represent knowledge in frame-like structures around the functions of the components. Using a pre-established model of the domain, components are characterized manually in a more or less subjective process. As the libraries grow much effort is required to index new components and to maintain the integrity of the indexing system. For example in GRANT [Cohen and Kjeldsen, 87] , an expert system for finding funding sources given research proposals, the knowledge base is a semantic network of research topics representing the research interests of 700 funding agencies. Roughly four person-months of effort, with the help of an expert funding adviser, were required to build GRANT's 4,500 node, 700-agency network.

Similar to the above systems, active browsing aims at assisting the user in the task of software retrieval. However active browsing makes use of the intrinsic relationships of object oriented software libraries and the code browsers in-built mechanisms for navigating such relationships. As new code is created they are automatically added to the library. On the other hand the KB approach requires constant maintenance of the library as new software components are added to the library.

7.1.4 Other Approaches

All automated software retrieval systems presented so far are based on text-retrieval methods. However, software is different from text in that it is executable. [Podgurski & Pierce, 93] proposed a new method for automated retrieval of software

components, called behavior sampling, that exploits this distinctive property. Behavior sampling identifies relevant routines by executing candidates that match a given set of inputs (supplied by the user) and by comparing their output to the output provided by the user. One difficulty with the method is the problem of determining the output of a component. In some applications this may be difficult to measure, e.g. output of a compiler. Moreover, this method is not applicable to non-executable software components such as design and documentation.

7.1.5 Browsing as a method of locating software

The type of the dialogue between the system and the user is as important (if not more) as the type of retrieval mechanism the system uses. In our systems, browsing is one form of interaction between the library and the user.

Browsing is more applicable for software libraries than for other kinds of libraries, since there rarely exists a component perfectly matching a user's query. This is particularly true when there is no good goal definition. Maarek agrees that browsing is helpful even in text-retrieval systems : "Browsing allows the user to discover unanticipated opportunities for reuse.." [Maarek et al., 91] (p. 800).

Hypertext systems are based on the browsing paradigm. Kiosk [Creech et al., 91] is a prototype that uses hypertext to enhance the process of component selection. Kiosk consists of a set of tools that can create , browse, and modify nodes and links in a software library. Kiosk uses graphical views of the library that help the user to better understand the contents of the libraries, hence more effectively selecting workproducts. Although the structural view of the library was found to aid searching, nonetheless, it was found that when first searching the library users get lost in the hypertext web.

The benefit of browsing is further shown in a study by [Egan et al., 89] in which students using a statistic textbook via a hypertext interface, perform better in a test than students using conventional printed text. The hypertext browser used is Superbook [Remde et al., 87]. It takes as input one or more existing documents in a standard text formatting language and presents the text in a multiwindow display with search and navigation enhancements. "SAM" [Perlman, 87] and the "Document Examiner" [Walker,

87] are similar browsing systems. Browsing is particularly suited to object oriented languages. Control flow graphs are browsable through intrinsic class relationships. Standard code browsers provide such functionalities through automatic cross referencing of source code identifiers. Contrary to the KB approach the indexing of class items is simple. However users require assistance during browsing to avoid getting lost.

7.2 Personal assistants

Our system learns automatically all its information by monitoring the user's browsing actions. In Machine Learning, such a system is called a learning apprentice. [Mitchell et al., 85] define the characteristics of Learning apprentices as "the class of knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem solving steps contributed by their normal use of the system" (p. 573). The system can provide assistance to the user in different ways: by predicting the user's next action, by adapting its response to the user's specific needs and experience, or by critiquing or correcting the user's input or activity.

A common feature systems discussed in this section is that they receive feedback from the user indicating the correctness of their predictions. Although in some cases, such as Eager [Cypher, 91], the form of the feedback may be minimal, it is still intrusive and distracts the user from the task at hand.

7.2.1 Predictive Interfaces

[Maes and Kozierok, 93] identifies three learning opportunities for a software agent: observing the user's action and imitating them, receiving user feedback upon error, and incorporating explicit training by the user. This framework is applied in simple learning apprentices such as a meeting scheduler [Kozierok and Maes, 93] and a personalized news system [Sheth and Maes, 93].

A semi-intelligent agent assists the user in the task of scheduling meetings. The agent uses simple but powerful learning techniques: Memory-based learning and Reinforcement Learning. This is similar in form to the work by [Dent et al., 92] except different learning methods are used. In both cases the systems predict additional features

of the meeting from those a user has already entered. Maes's system can also autonomously set up meetings by interacting with calendar programs of other attendees. In further work by Maes [Sheth and Maes, 93], personalized agents perform information filtering and suggest USENET news articles that might be of interest to the user. Agents that are successful in predicting items of interest to the user are assigned a higher fitness value and a form of artificial evolution is used to improve overall performance.

[Schlimmer and Hermens, 93] describes an interactive note-taking system for pen-based computers to speed up information entry and reduce user errors. The system actively predicts what the user is going to write and automatically constructs a custom graphical interface at the user's request. As in the above systems, a machine learning component characterizes the syntax and semantics of the user's information and this learned information is used to generate completion strings.

Feedback is crucial to the operation of these systems. Feedback provides new, highly informative data that can be used to improve the agent's subsequent predictions. In the news reading application, the user indicates immediately whether the articles retrieved are relevant or not. In Maes' meeting scheduler, the agent learns by reinforcement learning, relying on direct user's feedback.

These systems are similar to active browsing in that they draw inferences by monitoring user actions. However these systems do not necessarily infer the user's goal. Most personal assistants assist the user by predicting the user's next action or input. In contrast, in our system, the "learning agent" suggests to the user the object believed to be the user's search goal. In addition active browsing does not require user feedback to improve the system's inference.

7.2.2 Intelligent Interfaces that adapt to a specific user

Adaptive interfaces concentrate largely on adapting to the user in order to minimize user error, minimize the need for the user to request help, or speed up the process by anticipating the user's needs. The learning agent get its input from user interaction with the system, but contrary to predictive systems, the result of the agent's inference is to build a user model. The user model is a profile of the user, based on which

the agent provides more appropriate responses. The two approaches discussed here apply user modeling to information retrieval.

ARGOS [Bueanaga et al., 92] enhances the UNIX help system by building a user model so as to customize help to the user. ARGOS makes use of existing UNIX man pages and facilitates the access of the user to the right information at the right time. UNIX man pages are classified based on Maarek's lexical affinities. The User Model, UM, component of ARGOS holds general information about the user: his previous queries, and lists of relevant and irrelevant documents. The UM tries to guess the user's interests in order to modify the subsequent searches or to make corrections to the output of the information retrieval component. Information about the user is added incrementally by means of interaction with the user. Similar to Selective Search, the UM focuses the search using short term memory of the user's interest. However, the user can reset search or interest by use of a reset button.

[Jennings and Higuchi, 93]'s personal USENET news service proposes a user model for browsing, where the user is uncertain of exactly which information he desires. Contrary to ARGOS, the user model is built from session to session to accumulate a long term user model. The user model, a neural network that can be constructed incrementally, attempts to model the user's interests in terms of words and their frequency of appearance in articles the user reads. Retrieval is by spreading activation in the user model network. Articles are ranked by the sum of the energy of all active nodes in the user model. HYPERFLEX [Kaplan et al., 93], is another example of an adaptive system that learns user's preferences captured in associative matrices and recommends hypertextual information to users.

Maintaining a user model over the long term may be appropriate in the news service application. What the agent learns about the user would normally hold over many use sessions. In the retrieval task of software reuse, users typically solve different problems at different sessions. In our browsing model the interest of the user can shift completely from one session to another. Thus there is no need to build a long term model of the user.

In general adaptive systems are similar to ours in that no feedback is required from the user to validate the user model. Asking users to provide relevance feedback for particular features can be useful when it is unclear how to infer these features ratings from more general ratings, or when users are willing and able to do extra work involved in providing detailed relevance feedback. In personal assistants, it provides more reliable feedback than automatic learning. For instance, in information retrieval systems, it is demonstrated that relevance feedback provide significant increases in performance[Haines and Croft, 93; Turtle and Croft, 90]. [Belew, 89]'s Adaptive Information Retrieval (AIR) system requires users to rate explicitly documents suggested by the system. AIR uses feedback to slightly alter the representation of the documents, so that the system can learn from experience.

However, relevance feedback requires some extra effort on the part of the user. Asking users to provide relevance feedback increases the cognitive load on the user and may distract the user on the task at hand. The use of feedback can be perceived as a design trade-off between accuracy and the intrusiveness of the system. Moreover, "In some cases users cannot provide detailed relevance feedback... the users may not be aware of all the features that influence their preferences ...." [Kaplan et al., 93] (p. 200).

In contrast, [Kok and Botman, 88]'s Active Data Base makes inferences about users' preferences without requiring users to use a rating scale. The system infers similar car attributes from the choices users make when shown a list of automobiles. The inference is used to show users cars with similar attributes. This is perfectly non-intrusive: users are asked to behave in much the same way they would if shopping for a car.

7.2.3 Critiquing

Another approach to provide intelligent assistance is "Critics". Critiquing is a way to present a reasoned opinion about a product or action. Although most critics make suggestions on how to improve the product, the core task of critics is the recognition of deficiencies in a product and communication of those observations to users. Critiquing is an effective approach to make use of knowledge bases (KB) to aid users in their work and to support learning.

JANUS [Fischer and Giergensohn, 90; Fischer et al., 90] is a design environment based on the critiquing approach that allows designers to design residential kitchen. JANUS contains knowledge about how to distinguish "good" and "bad" designs, and can explain that knowledge. Critics in JANUS apply this design knowledge to critique the designer's partial solutions. Interestingly, JANUS can critique bad examples, thus allowing users to learn from negative examples.

Critics are also a major component of Terveen's *collaborative manipulation* [Terveen et al., 91], an approach to cooperative system design. In this context, the *HITS Knowledge Editor, or HKE,* tool was developed to assist users in the task of knowledge editing - the design, entry, viewing, and modification of structures in a KB. In HKE, critics examine the state of partially designed structures in a "workspace" and provides three types of assistance: infer required information not specified by the user, detect inconsistencies between information provided by the user and information in the KB, and suggest additional representational issues for users to consider. Studies showed that knowledge editing expertise embodied in HKE facilitated the performance of novice users.

The critic approach is attractive in that it has generality across a wide range of domains. [Fischer et al., 90] provides an overview of critiquing systems and describes a few successful commercial applications developed for such varied domains.

Although most critics make suggestions on how to improve the product, critics do not necessarily solve problems for users. Their main task is the recognizing deficiencies and communicating them to the users. On the other hand in the browsing task, our goal is not to teach the user how to find the target, but to actually identify it. As such, teaching the user how to browse better would not help him/her to find the target. However this can be achieved by familiarizing the user with the library semantics and structure.

Similar to active browsing, critics monitor user interactions with the system and present advice to users. However the critiquing approach is highly intrusive, involving an active interaction between the user and the critic. The user is required to provide inputs or feedback on the critic inferences by providing inputs, which are outside the normal interaction with the system. Furthermore various aspects of the interaction between the user and the system makes determining an non-intrusive format for presenting advice a

complex problem. In active browsing this problem is reduced to the simple task of rank ordering a list of suggestions.

7.2.4 Programming by Demonstrations

Programming by Demonstration is a method that allows end users to create, customize, and extend programs by demonstrating what the program should do [Cypher, 90]. Contrary to "Personal assistants" where the assistant acts as a teacher, PBD systems learn from the user's actions and make generalizations about high-level events, allowing them to automate complex tasks or take over the user's activity.

User feedback serves to let the user verify the system's inferences, and to obtain guidance from the user as to the salient features of the example over which to generalize. Similar to personal assistants, user feedback is an essential feature of PBD systems. This feedback may take many forms: e.g. in Peridot [Myers, 90] and Metamouse [Maulsby and Witten, 89], the system queries the user about every inference and requires Yes/No responses.

In general, most PBD systems generalize across multiple examples. While negative examples have been very useful in machine learning, they are problematic for demonstrations. It is difficult to demonstrate, for instance, that an object was selected because it is not red. TELS [Mo, 89] and Eager [Cypher, 91] make use of negative examples in a very limited way: these systems make predictions, and if the user rejects the prediction, they treat the prediction as a negative example. However this approach requires user 's feedback and hence is intrusive.

In [Witten, 81]'s Predictive Calculator, the system infers iterative computations from an initial sequence of keys. It predicts the next key sequence by the systematic processing of a "history list" of previous interactions. The systems rely on user's feedback to correct its predictions. For the next iteration, the system displays its guess but gives the user the choice to accept or reject. The Reactive Keyboard [Darragh and Witten, 92], a device designed to accelerate typewritten communication with a computer by predicting the user's next key, is based on the same idea of a history list.

A demonstrational interface with inductive inference capabilities - EDWARD [Bos, 92] is an action-inferring interface that predicts future actions based on the actions the user performs in a filemanager. It generalizes over arguments and results and detects patterns on the basis of a small sequence of user actions. EDWARD constantly monitors the user's actions and signals when it finds a repetition of actions. Similar to Eager, EDWARD's goal is to minimize intrusion of the user's actions. It uses a little icon to point at the item on which the next anticipated action is inferred.

Similar to PBD systems the active browser learns from the user browsing actions and tries to guess the user's search target. However a few essential differences exist between our system and PBD in general. Most PBDs predict the user next action but the active browser predicts the user's search target rather than the user's next browsing actions. Another difference is the use of user feedback to confirm the system's inference. Requiring user feedback in PBDs is intrusive and disrupts the user in the problem solving task. Active browsing makes use of the normal browser action and does not request user feedback in its inference.

7.3 Plan recognition systems

Plan recognition is the technique of inferring a user's plan from a given input, usually the user's actions. Researchers believe that plan recognition is an important issue in Human-Computer Interaction, and argue that plan inference is both useful and feasible in certain conditions.

[Goodman & Litman, 90], [Inoue and Nagata, 84] believe that by interpreting a user's actions with respect to a model of possible plans and goals, the interface is better able to provide a more intelligent class of system responses. The intelligent behavior can take several forms. In Planet [Quast, 93], the system provides relevant help topics in EXCEL (Microsoft spreadsheet application). CHECS [Goodman and Litman, 90] is a Chemical Engineering CAD system that provides intelligent assistance in support tasks such as advice generation, task completion, context sensitive responses, and error detection and recovery. CHECS uses [Kautz, 87] plan recognition algorithm, one of the first to formally define plan recognition. [Inoue & Nagata, 84] propose an algorithm for

recognizing the plan of the user from sequences of the MS-DOS command lines. User adaptation as in [Hook et al., 93] give as an example the configuration of interactive systems to suit specific needs of specific users. The EUROHELP [Breuker et al., 87] project adopts the "buggy-model" approach, where no plan construction takes place. This approach bears the most similarity to active browsing. The system monitors the user's actions, constantly trying to infer the user's goal. The sequence of user actions is matched against the system's library of plans (correct, incorrect and non-optimal) in order to find one that fits.

Plan recognition is more versatile than active browsing as plans can be expressed for various domains. However as with knowledge based systems, the design of the library of plans is a major factor to the inference performance of a plan recognition system. A major design problem with plan recognition systems is library coverage. [Cohen et al., 90; Cohen and Spencer, 93] acknowledge that correctly specifying plan libraries is difficult. [McCalla et al., 92] propose a case-based approach to solve the coverage problem. Instead of matching models directly, patterns of previous model instantiations are matched against a case library.

This implies that such systems cannot be expected to infer all possible user goals, unlike in active browsing, where the user's goal is any library item which can be inferred by the active browser. Researchers [Goodman and Litman, 90] agree that plan recognition systems have little to say about the recognition of novel plans.

Chapter 8

8. Conclusions

This thesis has presented the methods of negative inference and selective search as ways of improving the speed and accuracy of active browsing. This should encourage reuse by reducing the problem of locating software in extensive libraries. Section 8.1 presents the conclusions drawn from this research. Section 8.2 discusses the limitations of our approach and the specific implementation, and also proposes ideas about how these limitations might be overcome. Section 8.3 identifies areas where selective search and negative inference can be extended and improved. Finally section 8.4 outlines the principal directions for continuing this research in the future.

8.1 Summary

In this thesis we have demonstrated that negative features can be inferred even when the user does not explicitly indicate that an item is not interesting. We identified basic conditions sufficient to make such inference with reasonable confidence and designed inference rules to implement negative inference in active browsing. We have also shown that a dramatic improvement of the active browser's performance can be obtained by 1) adding finer-grained features, in the form of subterms, to the system's representation of the search target, and 2) adding negative inference rules to the inference system. The active browser thus obtained is twice as effective at identifying the user's search target as the original, and it ranks the target much more accurately at all stages of the search. More importantly, this gain is achieved while keeping the active browser unobtrusive and at no additional cost to the user since there is no change in the active browser interface.

Selective search was added to the active browsing system to speedup the matching process. We have shown that selective search can save considerable time with little loss in inference performance. Empirical results have shown that selective search can achieve similar inference performance as exhaustive search when evaluating roughly a quarter of the library. It was also observed that when evaluating less than one-eighth of the library

active browsing is ineffective at inferring the search target. It should be noted that these values pertained to the particular software library on which the system was tested.

8.2 Limitations

The results and conclusions presented in this thesis were obtained in experiments using an automated search agent. There is no doubt that the active browser should be tested in real-life situation with human users. Conducting large scale experiments involving human users was considered impractical given the scope of this thesis. However the search strategy adopted for the automated search agent is not unreasonable. In a limited study of real-life usage[Drummond et al., 96], it was found that some human users follow similar search strategy.

The inference rules are not generic, they are very specific and dependent on the browser interface. The negative inference rule identifies a particular user search pattern and would only generate negative features for users that follow this pattern. But in this browser this pattern is quite a good one, and is used by human users quite effectively. Moreover in some cases the performance gain of negative inference may not be realized. Empirical results shows that in few cases the Base active browser finds the search target faster than the negative active browser by a couple of steps. However this only happened rarely.

Selective search is a time saving process. One key factor to the efficiency of selective search is the low overhead of the selective search process. However this requires the partial score and the ordering of all items in the library to be maintained in memory. This puts a limit to the size of the library that can be managed internally by selective search. However the memory overhead for selective search is minimal as only the scores are saved.

8.3 Extending Selective search and Negative inference

In our implementation of selective search, we used a migration policy in which the same number of items migrate per partially scored set (PSS). This migration strategy does not make use of any knowledge about the PSS. A better method would be to assign

different migration levels to different PSS's based on a measure of which PSS is the "best" candidate for migration. The "best" PSS could be determined by the g-values and h-values of its items.

The sum of unscored template predicates (h-value) of a PSS can be used to assess the suitability of a PSS for migration. For example, consider three consecutive PSS's , p1, p2 and p3, where p1 has the most template terms evaluated (lowest uncertainty). A greater difference of h-value between consecutive PSS's indicates a greater potential to obtain high scores after migration. If the difference in h-value between p1 and p2 is higher than that between p2 and p3, then more items from p2 could migrate into p1 than p3 into p2.

Another alternative is to use the exact score of partially evaluated items (g-value). The g-value of the last item in a PSS can function as a threshold for determining which items from the lower PSS can migrate. Items in the lower PSS having a higher g-value than the last one in a PSS have scored higher on less templates evaluated. After migration to the next higher PSS they will invariably rank higher than the last item in the higher PSS.

The method of negative inference includes pruning the analogue subterm predicates on negative analogue predicates, but matching uses only positive features. In essence, negative inference is used to refine the analogue. The matching process could be made richer and more effective if negative scores were assigned to items containing negative inferred predicates. This may result in faster and more informative search, although with the added overhead of scoring on more template predicates.

The method of inferring negative features can be extended to other user browsing actions or patterns. We identified a user search pattern where negative inference can be applied. In this pattern the user selects items from a list and applies a specific operator on the selected items. Other such patterns or two stage operators could be identified by studying users of the browsing interface in real-life situations.

8.4 Future Work

In the short term, the main work is to refine the selective search and negative inference methods as described in the previous section. Improvements could be tested with the existing automated search agent. Another approach is to select a few human subjects as search agents and evaluate the active browser in real-life use. As was suggested in section 6.4, the selectivity of an active browser can be tuned while the browsing session is in progress. Extensive experiments and analysis is required to determine the optimal selectivity at various points in the search.

Although the development of selective search was motivated by active browsing, we can easily see applications of selective search to general search problems. Applications such as library database search and world wide web browsing are good candidates for selective search and active browsing. The size of the library database and the growth of the web makes searching a long process. The user interface or browsing interface should provide diverse opportunities for enhancing goal inference. For example in web browsers users can keep bookmarks of interesting web pages. The bookmarks can be used to initialize the analogue at the start of a browsing session or to increase the bias on analogue terms that match bookmarked items.

Selective search is conceptually similar to a constrained spreading activation search in graphs. By generalizing selective search to search problems where spreading activation is used, the applicability of selective search can be extended to other domains. One can easily imagine numerous cpu intensive applications that would benefit from selective search.

9. Bibliography

[Bates, 86]

Bates, M.J., "Terminological Assistance For the Online Subject Searcher", Proceedings of The Second Conference on Computer Interfaces For Informational Retrieval, Alexandria, Virginia, 1986, pp. 7.

[Belew, 89]

Belew, R.K., "Adaptive Information Retrieval: Using Connectionist Representation to Retrieve and Learn About Documents", Proceedings of SIGIR, Cambridge, MA., 1989, pp. 11-20.

[Biggerstaff and Richter, 87]

Biggerstaff, T. and C. Richter, "Reusability framework, assessment and directions". In Frontier Series: Software Reusability: Volume 1-Concepts and Models, ACM Press, New York, 1987, pp. 1-17.

[Bos, 92]

Bos, E., "Some Virtues and Limitations Of Action Inferring Interfaces", 5th Annual Symposium on User Interface Software and Technology, 1992, pp. 79-88.

[Breuker et al., 87]

Breuker, J., R. Winkels and J. Sandberg, "A Shell for Intelligent Help System", in International Joint Conference on Artificial Intelligence-87, 1987, pp. 167-173.

[Brownston et al., 85]

Brownston, L., R. Farrell, E. Kant and N. Martin, "Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming". Reading, MA: Addison-Wesley, 1985.

[Buenaga et al., 92]

Buenaga, M., B. Fernandez, A. Vaquero and A. Urech, "An On Line Intelligent Assistant for UNIX", Tech. Report DIA 92/16. Department of Computer Science, Complutense University of Madrid, 1992.

[Cohen et al., 90]

Cohen, R., F. Song, B. Spencer and P.V. Beek, "Plan Recognition beyond fixed Plan Libraries", in Proceedings of the Second International Workshop on User Modeling, Honolulu, HI, March 1990.

[Cohen and Kjeldsen, 87]

Cohen, R. and R. Kjeldsen, "Information Retrieval by Constrained Spreading Activation in Semantic Networks", Information Processing and Management, Vol 23, no. 4, 1987, pp. 255-268.

[Cohen and Spencer, 93]

Cohen, R. and B. Spencer, "Specifying and Updating Plan Libraries for Plan Recognition Tasks", Proceedings of the 9th Conference on Artificial Intelligence Applications, Monterey, California, March 2-6, 1993, pp. 27-33.

[Creech et al., 91]
Creech, M.L., D.F. Freeze and M.L. Griss, "Using Hypertext In Selecting Reusable Software Components", 3rd ACM Conference on Hypertext Proceedings, 1991, pp. 25-38.

[Cypher, 90]
Cypher, A., Ed., "Watch What I Do: Programming by Demonstration". MIT Press, Cambridge, MA, 1990.

[Cypher, 91]
Cypher, A., "Eager: Programming Repetitive Tasks by Example", Proceedings of CHI '91 conference proceedings, New Orleans, Louisiana, April 27-May 1991, pp. 33-39.

[Darragh and Witten, 92]
Darragh, J. and I. Witten, "The Reactive Keyboard", Cambridge University Press, New York, 1992.

[Dent et al., 92]
Dent, L., J. Boicario, J. McDermott, T. Mitchell and D. Zabowski, "A Personal Learning Apprentice", Proceedings of AAAI, 1992, pp. 96-103.

[Deutsch, 89]
Deutsch, I. P., "Design reuse and frameworks in the Smalltalk-80 system". In Frontier Series: Software Reusability: Volume II-Applications and Experience. Biggerstaff, T.J. and Perlis A.J. eds, ACM Press, New York, 1989, pp. 57-71.

[Devanbu et al., 91]
Devandu, P., R.J. Brachman, P.G. Selfridge and B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System", Communications of the ACM, May 1991, Vol. 34, no. 5, pp. 34-49.

[Drummond, 92]
Drummond, C., "Automatic Goal Extraction from User Actions to Accelerate the Browsing of Software Libraries", M.A.Sc. Thesis, Dept. of Electrical Engineering, University of Ottawa, December 1992.

[Drummond et al., 96]
Drummond, C., D. Ionescu and R.C. Holte, "A Learning Agent that Assists the Browsing of Software Libraries", Dept. of Computer Science, University of Ottawa, (Unpublished Report), 1996.

[Duchier, 93a]
Duchier, D., "Implementing search strategies with token passing graphs", University of Ottawa, Department of Computer Science, (Unpublished Report), 1993.

[D chier, 93b]
Duchier, D., "Concrete Browsing Of A Graphical Toolkit Library", University of Ottawa, Department of Computer Science, (Unpublished Report), 1993.

[Egan et al., 89]
Egan, D., J.R. Remde, T.K. Landauer, C.C. Locahbaum, and L.M. Gomez, "Behavioral Evaluation and Analysis of a Hypertext Browser", CHI '89 Proceedings, May 1989, pp. 205-210.

[Fernandez-Chamizo et al., 93]
Fernandez-Chamizo, C., L. Hernandez-Yanez, P.A. Gonzalez-Calero and A.U. Braque, "A Case-Based Approach to Software Component Retrieval," Symposium on Case-Based Reasoning and Information Retrieval, Stanford University, AAAI Spring Symposium Series, March 1993, pp. 35-44.

[Fischer et al., 90]
Fischer, G., A.C. Lemke, T. Mastaglio and A.I. Morch, "Using Critics to Empower Users", Empowering People: CHI'90 Conference Proceedings, April 1990, pp. 337-347.

[Fischer and Girgensohn, 90]
Fischer, G. and A. Girgensohn, "End-User Modifiability in Design Environments", Empowering People: CHI'90 Conference Proceedings, April 1990, pp. 183-191.

[Fouqué and Matwin, 92]
Fouqué, G., and S. Matwin, "CAESAR: A System for Case-based Software Reuse", Proceedings of Knowledge-Based Software Engineering-7, 1992, pp. 90-99.

[Frakes and Nejmeh, 87]
Frakes, W.B. and B.A. Nejmeh, "Software Reuse through Information Retrieval", Proceedings of the 12th Annual Hawaii International Conference on System Sciences, Kona, HI, 1987, pp. 530-535.

[Fraser et al., 89]
Fraser, S.D., J.M. Duran and R Aubin, "Software Indexing For Reuse", Proceeding of 1989 IEEE International Conference on Systems, Man and Cybernetics, 1989, pp. 853-858.

[Freeman, 83]
Freeman, P., "Reusable software engineering: Concepts and Research Directions". In Workshop on Reusability in Programming. ITT Programming, Stratford, Conn., 1983, pp 2-16.

[Goodman and Litman, 90]
Goodman, B.A. and D.J. Litman, "Plan Recognition for Intelligent Interfaces", Proceedings of the 6th Conference on Artificial Intelligence Applications, Santa Barbara, CA, 1990, pp. 297-303.

[Haines and Croft, 93]
Haines, D. and W.B. Croft, "Relevance Feedback and Inference Networks", Proceedings of 16th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval, 1993, pp. 2-11.

[Helm and Maarek, 91]
Helm, R. and Y.S. Maarek, "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries". Proceedings of OOPSLA-91, 1991, pp. 47-60.

[Henninger, 93]
Henninger, S.R., "Locating Relevant Examples for Example-Based Software Design", PhD. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1993.

[Holte and Ng, 96]
Holte, R.C. and J. Ng Yuen Yan, "Inferring What a User is Not Interested in". Advances in Artificial Intelligence/AI '96, Lecture Notes in Artificial Intelligence, no. 1081, Springer, 1996, pp. 159-171.

[Hook et al., 93]
Hook, K., J. Kerlgren and A. Woern, "Inferring Complex Plans", Proceedings of the 1993 International Workshop on Intelligent User Interfaces, 1993, pp. 231-234.

[Inoue and Nagata, 84]
Inoue, Y. and M. Nagata, "A Case Study of Plan Recognition -- Command Sequences as Acts --", Proceedings of the 1st U.S.A. - Japan Conference on Human Computer Interaction, Honolulu, Hawaii, August 18-20, 1984, pp. 392-397.

[Jennings and Higuchi, 93]
Jennings, A. and H. Higuchi, "A User Model Neural Network for a Personal News Service", User Modeling and User-Adapted Interaction, vol. 3, 1993, pp. 1-25.

[Kaplan et al., 93]
Kaplan, C.A., Fenwick and J., Chen, J., "Adaptive Hypertext Navigation Based On User Goals and Context", User Modeling and User-Adapted Interaction, vol. 3, 1993, pp. 193-220.

[Kautz, 87]
Kautz, H.A., "A Formal Theory of Plan Recognition", PhD thesis, University of Rochester, N.Y., 1987.

[Kok and Botman, 88]
Kok, A.J. and A.M. Botman, "Retrieval Based on User Behavior", Proceedings of the 11th International Conference on Research and Development in Information Retrieval, Grenoble, 1988, pp. 343-358.

[Kozierok and Maes, 93]
Kozierok, R. and P. Maes, "A Learning Interface Agent for Scheduling Meeting", Intelligent User Interfaces '93, 1993, pp. 81-88.

[Krueger, 92]
Krueger, C.W., "Software Reuse", ACM Computing Surveys, Vol. 24, No. 2, 1992, pp. 131-184.

[Lang, 95]
Lang, K., "NewsWeeder: Learning to Filter Netnews', Proceedings of the 12th International Conference on Machine Learning, Morgan Kaufmann, 1995, pp. 331-339.

[Liskov, 87]
Liskov, B., "Data abstraction and hierarchy", In OOPSLA '87, Addendum to the Proceedings, ACM Sigplan, New York, 1987, pp. 17-34.

[Maarek et al., 91]
Maarek, Y.S., D.M. Berry and G.E. Kaiser, "An Information Retrieval Approach For Automatically Constructing Software Libraries". IEEE Transactions On Software Engineering, Vol 17 no 8, Aug. 1991, pp. 800-813.

[Maes and Kozierok, 93]
Maes, P. and R. Kozierok, "Learning Interface agents", Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI Press, 1993, pp. 459-465.

[Matwin and Ahmad, 94]
Matwin, S. and Ahmad, A., "Reuse of Modular Software with Automated Comment Analysis", Proceedings of Intl Conference on Software Maintenance '94, Victoria, BC, Sept. 1994, pp. 222-231.

[Maulsby and Witten, 89]
Maulsby, D. and I. Witten, "Teaching a Mouse How to Draw", Proceedings of Graphics Interface '89, London, Ontario, June 1989, pp. 130-137.

[McCalla et al., 92]
McCalla, G., J. Greer and R. Coulman, "Enhancing the Robustness of Model-Based Recognition", in Proceedings of 3rd International Workshop on User Modeling, Aug. 1992, pp. 240-248.

[Mitchell at al., 85]
Mitchell, T.M., Mahadevan, S., and L.Steinberg. "LEAP: A Learning Apprentice for VLSI Design". Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1985, pp. 573-580.

[Mo, 89]
Mo, D., "Learning Text Editing Procedures from Examples", M.Sc. Thesis, Dept. of Computer Science, University of Calgary, 1989.

[Myers, 90]
Myers, B., "Creating User Interfaces Using Programming by Example", Visual Programming and Constraints, ACM Transactions on Programming Languages and Systems, Vol 12, No. 2, 1990, pp. 143-177.

[Perlman, 87]
Perlman, G., "An overview of SAM: A hypertext interface to Smith & Mosier's". Technical Report TR-87-09. Wang Institute, Tyngsboro, Ma, 1987.

[Podgurski and Pierce, 93]
Podgurski, A. and L. Pierce, "Retrieving Reusable Software by Sampling Behavior", ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 3, 1993, pp. 286-303.

[Prieto-Diaz, 91]
Prieto-Diaz, R., "Implementing Faceted Classification For Software Reuse". Communications of the ACM, Vol 34, 1991, pp. 89-97.

[Quast, 93]
Quast, K.J., "Plan Recognition For Context Sensitive Help", Proceedings of the 1993 International Workshop on Intelligent User Interfaces, 1993, pp. 89-93.

[Remde et al., 87]
Remde, J.R., L.M. Gomez and T.K. Landauer, "Superbook: An Automatic Tool for Information Exploration--Hypertext", In Proceedings of Hypertext '87. University of North Carolina, Chapel Hill, 1987, pp. 175-188.

[Schlimmer and Hermens, 93]
Schlimmer, J.C. and L.A. Hermens, "Software Agents: Completing Patterns and Constructing User Interfaces", Journal of Artificial Intelligence Research 1, 1993, pp. 61-89.

[Sheth and Maes, 93]
Sheth, B. and P. Maes, "Evolving Agents for Personalized Information Filtering", Proceeding of the 9th Conference on Artificial Intelligence For Applications, 1993, pp. 345-352.

[Terveen et al., 91]
Terveen, L.G., D.A. Wroblewski and S.N. Tighe, "Intelligent Assistance through Collaborative Manipulation", IJCAI-91: Proceedings of the 12th International Conference on Artificial Intelligence, Sydney, Australia, 1991, pp. 74-79.

[Turtle and Croft, 90]
Turtle, H.R. and W.B. Croft, "Inference Networks for document retrieval", In Proceedings of the 13th International Conference on Research and Development in Information Retrieval, Brussels, Belgium, 1990, SIGIR, pp. 1-24.

[Walker, 87]
Walker, J.H. "Document Examiner: Delivery interface for hypertext documents". In Proceeding of Hypertext '87. University of North Carolina, Chapel Hill, 1987, pp. 307-323.

[Witten, 81]
Witten, I.H., "Programming by example for the casual user: a case study," Proceedings Canadian Man-Computer Communication Conference, Waterloo, Ontario, 1981, pp. 105-113.

[Wong, 92]
Wong, Y.C., "Using Version Spaces to Support Incremental Searches in a Software Library," in Proceedings of the 4th International Conference on Software

Engineering and Knowledge Engineering, IEEE Comp. Society, 1992, pp. 412-419.

Appendix A: Analogue Inference Rules

```
;;;
;;; Browsing action predicate
;;;
(literalize Browsing-action cur-node-type cur-node-name op pr-node pr-op cf)

;;;
;;; analogue predicates
;;;
(literalize InterestedIn type name new cf)
(literalize NotInterestedIn type name new cf)
(literalize ChildrenOf Class name new cf)
(literalize SiblingOf  class pr-node name new    cf)
(literalize AClassInh name)
(literalize tried sel name scale)

;;;
;;; Analogue updating rules
;;;

(p rule1
  {<Incoming-Ba>
    (Browsing-action ^cur-node-type class ^cur-node-name <cur-name> ^pr-node <name> ^pr-op sub)}
  {<child>
   (ChildrenOf ^Class class ^name <name> ^new old ^cf <cf>)}
  - (InterestedIn ^type class ^name <cur-name>)
  -->
  (remove <Incoming-Ba>)
  (make AclassInh ^name <name>)
  (make InterestedIn ^type class ^name <cur-name> ^new new ^cf 0.01)
  (make ChildrenOf ^Class class ^name <name> ^new new ^cf <cf>))

(p rule2
  {<Incoming-Ba>
   (Browsing-action ^cur-node-type class ^cur-node-name <cur-name>)}
-->
  (remove <Incoming-Ba>)
  (make InterestedIn ^type class ^name <cur-name> ^new new ^cf 0.01))

(p rule3
  { <Incoming-Ba>
   (Browsing-action ^cur-node-type class ^cur-node-name <cur-name> ^pr-node <name> ^pr-op sub)}
  - (ChildrenOf ^Class class ^name <name>)
  -->
  (remove <Incoming-Ba>)
  (make InterestedIn ^type class ^name <cur-name> ^new new ^cf 0.01)
  (make ChildrenOf ^Class class ^name <name> ^new new ^cf 0.02))

(p rule4
```

```
  {<Incoming-Ba>
   (Browsing-action ^cur-node-type class ^cur-node-name <cur-name> ^op sub ^pr-node <name> ^pr-op
sup)}
  {<Int-factor>
   (InterestedIn ^type class ^name <name> ^new old ^cf <cf>)}
  -->
  (remove <Incoming-Ba>)
  (make InterestedIn ^type class ^name <cur-name> ^new new ^cf 0.01)
  (make SiblingOf
    ^Class class ^pr-node <name> ^name <cur-name> ^new new ^cf <cf>))

(p rule5
  {<Incoming-Ba>
   (Browsing-action ^cur-node-type class ^cur-node-name <cur-name> ^op <op> ^pr-node <name> ^pr-
op sub)}
  {<sibling>
   (SiblingOf ^Class class ^pr-node <pname> ^name <name> ^new old ^cf <cf>)}
  -->
  (remove <Incoming-Ba>)
  (make AclassInh ^name <name>)
  (make InterestedIn ^type class ^name <cur-name> ^new new ^cf 0.01)
  (make
   SiblingOf ^Class class ^pr-node <pname> ^name <name> ^new new ^cf <cf>))

(p rule6
  {<Incoming-Ba>
   (Browsing-action ^cur-node-type class-type ^cur-node-name <cur-name> ^op SimCla)}
  -->
  (remove <Incoming-Ba>)
  (make tried ^sel SimCla ^name <cur-name> ^scale 0.5))

(p rule7
  {<Incoming-Ba>
   (Browsing-action ^cur-node-type class-type ^cur-node-name <cur-name> ^op SimNamCla)}
  -->
  (remove <Incoming-Ba>)
  (make tried ^sel SimNamCla ^name <cur-name> ^scale 0.5))

(p rule8
  {<Incoming-Ba>
  (Browsing-action ^cur-node-type method ^cur-node-name <mname>
        ^op << mark ImpClass source expansion selection >> ^pr-node <name>)}
  -->
  (remove <Incoming-Ba>)
  (make InterestedIn ^type method
        ^name <mname>
        ^new new
        ^cf 0.01)
  (make InterestedIn
     ^type class ^name <name> ^new new
        ^cf 0.005))

(p rule9
  {<Incoming-Ba>
```

```
  (Browsing-action ^cur-node-type subterm  ^cur-node-name <mname>
        ^op << mark ImpClass source expansion selection >> ^pr-node <name>)}
  -->
  (remove <Incoming-Ba>)
  (make InterestedIn ^type subterm ^name <mname> ^new new ^cf 0.01)
  (make InterestedIn ^type class
     ^name <name> ^new new
     ^cf 0.005)
)

(p rule10
  {<Incoming-Ba>
  (Browsing-action ^cur-node-type subterm ^cur-node-name <mname> ^op Unwanted-subterm)}
  - (NotInterestedIn ^type method ^name <mname>)
  -->
  (remove <Incoming-Ba>)
  (make NotInterestedIn ^type subterm ^name <mname> ^new new))

(p PruneSubterm
  (NotInterestedIn ^type subterm  ^name <mname> ^new old)
  {<NoisySubterm>
  (InterestedIn ^type subterm  ^new old ^name <mname>)}
  -->
  (remove <NoisySubterm>))

;;;
;;; Rules for updating analogue predicates confidence factor
;;;

(p rem-dup-int
  {<new>
   (InterestedIn ^type <type> ^name <name> ^cf <newcf> ^new new)}
  {<old>
   (InterestedIn ^type <type> ^name <name> ^cf <oldcf> ^new old)}
  -->
  (modify <old> ^cf (compute <oldcf> + ((1.0 - <oldcf>) * <newcf>)))
  (remove <new>))

(p mod-int
  {<new>
   (InterestedIn ^type <type> ^name <name> ^new new)}
  -(InterestedIn ^type <type> ^name <name> ^new old)
  -->
  (modify <new> ^new old))

 (p rem-dup-Notint
  {<new>
   (NotInterestedIn ^type <type> ^name <name> ^cf <newcf> ^new new)}
  {<old>
   (NotInterestedIn ^type <type> ^name <name> ^cf <oldcf> ^new old)}
  -->
 (remove <new>))

 (p mod-Notint
```

```
  {<new>
   (NotInterestedIn ^type <type> ^name <name> ^new new)}
  -(NotInterestedIn ^type <type> ^name <name> ^new old)
  -->
  (modify <new> ^new old))

(p rem-dup-child
  {<newChild>
   (ChildrenOf ^name <name> ^class <class> ^new new ^cf <newcf>)}
  {<old>
   (ChildrenOf ^name <name> ^class <class> ^new old ^cf <oldcf>)}
  -->
  (modify <old> ^cf (compute <oldcf> + ((1.0 - <oldcf>) * <newcf>)))
  (remove <newChild>))

(p mod-child
  {<newChild>
   (ChildrenOf ^name <name> ^class <class> ^new new )}
  -(ChildrenOf ^name <name> ^class <class> ^new old )
  -->
  (modify <newChild> ^new old))

(p rem-dup-sib
  {<newSib>
        (SiblingOf  ^pr-node <pnode> ^name <name> ^class <class> ^new new ^cf <newcf>)}
  {<old>
        (SiblingOf  ^pr-node <pnode> ^name <name> ^class <class> ^new old ^cf <oldcf>)}
  -->
  (modify <old> ^cf (compute <oldcf> + ((1.0 - <oldcf>) * <newcf>) ))
  (remove <newSib>))

(p mod-Sib
  {<newSib>
   (SiblingOf
        ^pr-node <pnode> ^name <name> ^class <class> ^new new )}
  -(SiblingOf
        ^pr-node <pnode> ^name <name> ^class <class> ^new old )
  -->
  (modify <newSib> ^new old))
```

Appendix B: Rules mapping analogue to template

```
;;;
;;; template predicate
;;;
(literalize Template type name new scale cf time weight)

;;;
;;; Rules for mapping analogue predicates to template predicates
;;;

(p Template-class
  (subtask ^name makeTemplate)
  (InterestedIn ^type class ^name <name> ^cf <cf>)
  -->
  (make Template ^type excludeClass ^name <name> ^scale 1.0 ^cf <cf>
        ^time curr ^new new ^weight (compute 1.0 * <cf>))
  (make Template ^type className ^name <name> ^scale 0.5 ^cf <cf>
        ^time curr ^new new ^weight (compute 0.5 * <cf>)))

(p Template-method
  (subtask ^name makeTemplate)
  (InterestedIn ^type method ^name <name> ^cf <cf>)
  -->
  (make Template ^type impMeths ^name <name> ^scale 0.5 ^cf <cf>
     ^time curr ^new new ^weight (compute 0.5 * <cf>)))

(p Template-method-subterm
  (subtask ^name makeTemplate)
  (InterestedIn ^type subterm ^name <name> ^cf <cf>)
  -->
  (make Template ^type subterm ^name <name> ^scale 0.5 ^cf <cf>
     ^time curr ^new new ^weight (compute 0.5 * <cf>)))

(p Template-classInhA1
  (subtask ^name makeTemplate)
  (ChildrenOf ^name <name> ^class <class> ^cf <cf>)
  (AclassInh ^name <name>)
-->
  (make Template ^type classInh ^name <name> ^scale 2.0 ^cf <cf>
        ^time curr ^new new ^weight (compute 2.0 * <cf>)))

(p Template-classInhB1
  (subtask ^name makeTemplate)
  (SiblingOf ^pr-node <pname> ^name <name> ^class <class> ^cf <cf>)
  (AclassInh ^name <name>)
-->
  (make Template ^type classInh ^name <name> ^scale 2.0 ^cf <cf>
     ^time curr ^new new ^weight (compute 2.0 * <cf>)))

(p rem-dup-temp
```

```
  (subtask ^name makeTemplate)
  {<new>
   (Template ^type classInh ^name <name>
         ^time curr ^scale <scale> ^cf <newcf> ^weight <nweight> ^new new)}
  {<old>
   (Template ^type classInh ^name <name>
     ^time curr ^scale <scale> ^cf <oldcf> ^weight <oweight> ^new old)}
  -->
  (remove <new>)
  (modify <old>
         ^cf (compute <oldcf> + ((1 - <oldcf>) * <newcf>))
         ^weight (compute <scale> * (<oldcf> + ((1 - <oldcf>) * <newcf>))) )

;;; this rule updates the confidence factor of two similar template predicate
(p mod-dup-temp
  (subtask ^name makeTemplate)
  {<new>
   (Template ^type classInh ^name <name> ^time curr
     ^scale <scale> ^cf <cf> ^weight <weight> ^new new)}
  - (Template ^type classInh ^name <name> ^time curr
     ^scale <scale> ^cf <cf> ^weight <weight> ^new old)
  -->
  (modify <new> ^new old))
```