An Information Retrieval Tool for Reverse Software Engineering

Christos Magdalinos
School of Computer Science
McGill University, Montreal

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of M.Sc. in Computer Science.

ISBN 0-612-19834-0

Name MAGDALINOS CHRISTOS

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

Computer Science
SUBJECT TERM

0	9	8	4

SUBJECT CODE

U·M·I

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

McGill University
FACULTY OF GRADUATE STUDIES AND RESEARCH

AUTHOR'S NAME: CHRISTOS MAGDALINOS

DEPARTMENT: Computer Science DEGREE SOUGHT: Master in Science

TITLE OF THESIS: An Information Retrieval Tool for Reverse Software Engineering

SHORT TITLE OF THESIS: ______

(70 chars. or less) ______

Authorization is hereby given to McGill University to make this thesis available to readers in the McGill University Library or other library, either in its present form or in reproduction. The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise be reproduced without the author's written permission. The authorization is to have effect on the date given below unless a deferral of one year is requested by the author on submitting the thesis.

Permanent Address of Author:
63 Asklipiou Street
166 73 Voula Athens Greece

Signature of Author: ______

Date: 25 June 1996

Abstract

Information retrieval in large data spaces using formal, structure oriented patterns of features has many possible applications. We developed and studied a system that can be used to localize code segments in a program. The system is built using a generic and extensible object oriented framework and uses the Viterbi dynamic programming algorithm on simple Markov models to calculate a similarity measure between an abstractly described code segment and a possible instantiation of it in the program. The resulting system can be incorporated in a larger cooperative environment of CASE tools and can be used during the design recovery process to perform concept localization.

Résumé

Le retrait d' information dans de grands espaces de données utilisant des modèles de traits formels et orientés structure a beacoup d' applications possibles. Nous avons developé et étudié un systeme qui peut être utilisé pour localiser des segments de code dans un programme. Le système est construit utilisant un structure générique et extensible orienté-objet, et utilise l' algorithm de programmation dynamique de Viterbi sur de simples modèles de Markov pour calculer une mesure de similarité entre un segment de code décrit abstraitement et son instantiation possible dans le programme. Le système resultant peut être incorporé dans un environment coopératif plus large d' outils CASE et peut être utilisé lors du processus de remise en marche du design pour performer la localisation de concepts.

Acknowledgements

First of all I would like to thank my thesis advisor, professor Renato De Mori, for his trust, support and advice during the period of this work.

I am also thankful to all the members of the speech lab for their help and co-operation. I have to explicitly express my gratitude to Charles Snow and Matteo Contolini for their advice in crucial times, their help during the testing phase of the system and most importantly their friendship.

I gratefully acknowledge the contribution of insightful ideas from all the people involved in the REVENGE project.

I am also grateful to my friends Luiza and Yiannis, for providing many happy distractions and constant encouragement.

Finally I would like to thank my friend and co-supervisor Kostas Kontogiannis for his continuous support, guidance and help. Without his valuabie insights and suggestions the completion of this work would not be possible.

I dedicate this work to my family and especially to my father for being a constant inspiration throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The time required to grasp the nature or the meaning of a newly created human artifact, given some description of it, grows with the artifact's complexity and the quality of its description. In our days human artifacts tend to be extremely complex, and although there might not be lack of information describing them, comprehending such products is always a difficult task.

Information systems, and computer programs specifically, are among the most intricate products one can come across today. To understand how such systems function one has to recapture the design and decipher the requirements actually satisfied and implemented by the subject system.

In order to comprehend how a program works three actions can be taken by an analyst: read about it (e.g. read documentation); inspect the source code or run it (e.g. watch execution, get trace data). Documentation is rarely excellent; in most cases it simply does not exist or is inadequate and misleading. Studying the dynamic behavior of an executing program can be useful but unfortunately is not always possible. That leaves the source code as the primary and sole trustworthy source of information. The investigation process which the analyst has to undertake is akin to *idea processing* [15]. The goal is to move from a chaotic collection of unrelated ideas to an integrated, orderly interpretation of these ideas and their interconnections.

Nowadays, one of the main obstacles for an analyst is the size of the source code. For successful systems, developed and enhanced through the years, the size is often expressed in millions of lines. The need for tools which can assist the analyst in this non trivial task is apparent.

This report describes our work creating a framework that can be used to build tools capable of retrieving information from large data spaces by comparing formal, structure oriented patterns of features; partial as well as complete matches are detected. The described framework was used to develop a system which focuses on source code for a specific programming language (namely C). The resulting system can be integrated in a larger cooperative reverse engineering environment (REVENGE [21]) consisting of various powerful CASE tools. A possible application of the system, when the input is source code in a programming language, is aiding software engineers to recapture and understand the design of a program.

1.1 Motivation

Program comprehension is an every day task for all programmers. Understanding a piece of code can be a critical subtask of debugging, modifying or simply getting familiar with a system. Reverse engineering is a supporting technology for program understanding and can be defined as the process of analyzing a subject system to :

- identify the systems components and their interrelationships,
- create representations of the system in another form at a higher level of abstraction [13].

The reverse engineering process involves extracting design artifacts and building or synthesizing abstractions that are less implementation dependent from a subject system: it is a process of examination not a process of replication or change [13, 62]. Systems that we do not know how to cope with but that are vital to an organization are called legacy systems [2]. Legacy systems represent years of accumulated experience and knowledge. Program understanding, and its subtask design recovery, become major maintenance activities when dealing with unstructured legacy systems.

Studies on how expert programmers remember code show they "chunk" code into meaningful program segments and then mentally organize the chunks based on the functional purpose of the code [65]. These chunks are often called *mental plans*, *clichés* or *concepts*. Concepts are implemented by pieces of code consisting of a set of program statements. We will refer to these pieces of code as *code segments*.

In other words the analyst uses his or her programming knowledge to recognize high level concepts. Typically this knowledge includes stereotyped code patterns of common programming strategies, data structures and algorithms. Using this heuristic-based knowledge the analyst skips trivial parts and looks only for things he deems important. As a result a functional model of the program is created and used to guide maintenance activities.

Capturing knowledge effectively for the maintenance task is an open theoretic problem. It is our belief that design recovery can not be fully automated. Whatever substitute for a human maintainer, during the design recovery process, has been proposed is simply not as effective. This observation led us to focus our research in creating tools capable of assisting the maintainer in his task interactively.

The system described in this document can be considered as a part of a hybrid design recovery system. Initially the analyst supplies an abstract description of a code segment, which implements a design concept, to the system which in turn, after exhaustive code analysis, returns all possible locations of this segment in the source code. Partial match is allowed and for every discovered location, a measure of the "distance" between the reported implementation of the concept and the segment description, is also calculated and reported. The maintainer can subsequently inspect the results and if necessary refine the concept's description and fine tune the system in order to achieve improved performance.

1.2 Goals and Objectives

The basic goal of this research is to evolve Ariadne, a prototype system built for the REVENGE project [21] which detects programming patterns. Ultimately the result of this effort is the creation of a generic framework which could be subsequently used to extend REVENGE. Therefore our system shares a number of common features with Ariadne, the most significant ones are:

- the same core algorithm using Markov Models and the Viterbi dynamic programming algorithm to calculate the best alignment between two code segments,
- the same schema for intermediate code representation,
- the capability of being integrated in the cooperative environment of CASE tools developed for the REVENGE project,
- a subset of the abstract language introduced in the prototype to describe code segments and
- it focuses on the same target language (C).

On the other hand the new system is significantly different from its predecessor in the following aspects:

- it is implemented in a different programming language using a new design,
- it is platform independent,
- it has a flexible and intuitive user interface,
- it uses different input source and representation,
- it is extensible and easy to maintain and finally

- its design can be reused to handle source code from different languages.

It was our belief that the algorithm introduced in the prototype could be the heart of a generic, reusable framework for information retrieval tools. Hence design recovery is one of possibly many other tasks (i.e. simple code localization, pattern matching based on a set of formally described features) depending on the target language, where the framework can be used. For this reason we consider the system as an *information retrieval* tool and not as a specialized design recovery tool. As a result the main objective of this work is the creation of such a generalized, reusable and extensible framework.

While building the system and writing this document the prototype build for the REVENGE project was still undergoing testing as well as significant changes and enhancements, because of this an evaluation based on quantitative or qualitative comparisons of the two tools was not possible. We do not claim to have built a better or more powerful system in respect to abstract language abilities, we can safely say though that the new system is more generic and flexible than its prototype.

The theoretical background, presented in chapter four, is essentially the one described in [38, 25, 26]. Presentation improvements of theoretical issues were made based on suggestions of the supervisors of this thesis.

The system built using the resulting framework focuses on *code segment localization* and was tested with several programs ranging from few hundred lines to several thousand lines. We were able to describe code segments implementing both generic and specific concepts and localize them in the code. During the experimentation phase we were also able to realize a number of possible improvements that are reported in the future work section in the chapter six.

1.3 Thesis outline

In the next chapter we elaborate on design recovery process issues and present related systems both commercial and experimental. At the end of the chapter there is a brief overview of the cooperative environment created for the REVENGE project. Chapters three and four contain detailed description of the system development process and its architecture. In chapter three we focus on system analysis issues and in chapter four on design and implementation issues. Chapter five presents our experimental results. Chapter six discusses ideas for future work and presents a summary of our conclusions. Finally appendix A presents a simplified description of the Abstract Concept Language (ACL) we use in Backus Normal Form (BNF), appendix B contains a few examples of concepts used in our experiments and appendix C presents a detailed example of concept localization using the described framework.

Chapter 2

Problem Description and Related Work

Much of the software used today in critical tasks is 10 to 15 years old [47]. Maintaining these usually successful systems involves a collection of puzzle-solving skills. It includes getting tools to do the software process right and being able to deal with unknown software and unmaintainable systems. Software maintenance practices account for fifty to ninety per cent of total life-cycle costs[13] and around two per cent of the gross national product in U.S according to a study published in 1990 [36].

Reverse engineering was the answer of the computer science community to the high demand for a systematic approach to solve such problems. Chikofsky and Cross in their influential work [13] adopt M.F.Rekoff's definition of reverse engineering as "the process of developing a set of specifications for a complex hardware system by an olderly examination of specimens of that system". The subject system is software and the objective is to gain sufficient design-level understanding to aid maintenance, strengthen enhancement or support replacement of the system.

We can divide reverse engineering in two major activities :

1. Redocumentation and

2. Design recovery.

Redocumentation is the process of creating alternative multiple views of the program in order to capture certain characteristics of the subject system. Design recovery focuses on creating abstractions in order to impose a "meaning" on a program segment.

Although there might be a slight disagreement in terminology it is widely accepted that reverse engineering is primarily a process of examination and not a process of changing or enhancing the subject system [13, 62]. The process of introducing new functionality or restructuring the subject system is called functional reengineering or simply reengineering.

Our system is a pure reverse engineering tool designed to aid the maintainer in his task to retrieve information in order to decipher designs from finished products. Later the analyst might of course use the acquired knowledge to reengineer the subject system while in the maintenance process. In the next sections of this chapter we will present the basic concepts in design recovery and work of other researchers in the field.

2.1 Design Recovery

Design recovery can be defined as a subset of reverse engineering in which domain knowledge, external info and deduction with a sort of fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself [13]. Biggerstaff adds that "design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience and general knowledge about problem and application domains ...". Using design recovery is some times the only way to salvage whatever we can from existing systems, it lets us get a handle of the system when we do not understand how they work or how their individual programs interact as a system.

The initial input for most design recovery systems is source code in an enhanced abstracted form. We call enhanced code, source code adorned with hints related to the code functionality. These adornments may have the form of comments, control and data flow information, annotations in the source code intermediate representation, I/O commands or just identation. Using this input the analyst should try to construct a higher level description of the program. The process is usually bottom up and incremental, the analyst detects low level constructs and replaces them with their high-level counterparts.

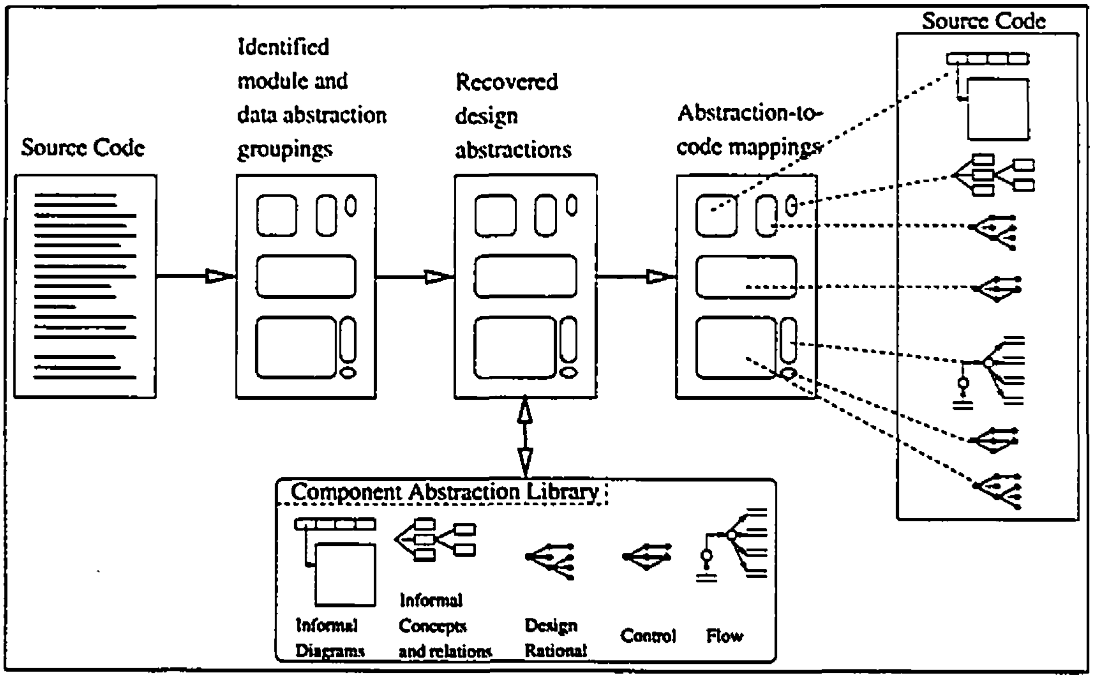

Source Code
Identified module and data abstraction groupings
Recovered design abstractions
Abstraction-to-code mappings
Source Code
Component Abstraction Library
Informal Diagrams
Informal Concepts and relations
Design Rational
Control
Flow

Figure 2.1: The design recovery process.

Given the actual program source code an analyst first looks for large-scale organizational structures such as the subsystem structure and important data structures. Useful design structures are also recovered and expressed in abstracted forms such as design rationale, module structures and informal diagrams,concepts and relations.

The next step in the process is the population of reuse and recovery libraries in order to facilitate further productive use of the recovered design components. In this step all recognized components go through a generalization process so they can be made available to a wider spectrum of applications. These generalized concepts are then stored in a library forming a domain model. Finally the abstract design components in the domain model become the starting point for discovering candidate realizations of themselves in a new system's code. These basic steps of the design recovery process are shown in figure 2.1.

The most common methods used in program understanding are data and control flow graph analysis.

Data flow analysis describes how information propagates from statement to statement and module to module. Control flow describes the sequence in which statements are executed and how control is passed from one module to the other. Usually the product of control flow analysis is a directed graph with annotations. Language analyzers are used to recognize language constructs which implement data flow [31].

The ability to view the subject system from different perspectives is one of the key objectives of reengineering [13]. An analyst can view the program from different levels of detail [30]:

1. the *implementation level view* abstracts away a program's language and implementation specific features, typically an Abstract Syntax Tree (AST) and a symbol table of program tokens are the produced artifacts,

2. the *structure level view* abstracts a program's language dependent details to reveal its structure from different perspectives, the result is an explicit representation of dependencies among program components,

3. the *function level view* relates pieces of the code to their functions to reveal the logical relations among them and finally

4. *the domain level view* further abstracts the function level view by replacing its algorithmic nature with concepts specific to the application domain.

All the resulting views are usually presented to the analyst as a graph. Graphs have been adopted as an intuitive and sound mathematical formalism to represent the structure of a computer program. Graph complexity can be a metric for the maintainability of the code. Prior experience using graphs in formal languages, compilers and parsers was used and several techniques were "ported" in the field [58, 10, 23].

2.1.1 Representation Methods

In order to move from the physical implementation of a system to high-level abstractions of its modules and the logical, implementation-independent, designs the analyst must ignore all unnecessary details embodied in the initial input. The following subsection examines some commonly used representation methods to achieve this task during the first step of the design recovery process (see figure 2.1).

The first task of the analyst is the creation of module and data abstractions. In this section we present some of the most important solutions proposed.

Several researchers chose to directly divide the code to: data and methods acting on the data, this is formally called the Data - Procedure code division. Describing data structures can be done using tabularization [63]. For each data structure we record its basic properties (i.e. name, position, type, length) in a table entry. Subsequent use of the resulting table as an input to transitive closure algorithms can compute data flow and variable dependencies [49]. By introducing Relationship Matrices the same technique can be used to capture relationships among procedures, constants and variables of procedures within the same module. One of the main advantages of this approach is that matrices can be stored as tables in any relational database. The analyst can then perform several queries on the stored data using advanced features that database environments offer.

Another way to abstractly represent source code is by mapping each basic language construct to an object and capture syntax as a list of attributes. This method was introduced Das in [18] and represents code as instances of the basic language construct classes. Automatic creation of objects, if a Backus Normal Form (BNF) description of the language exists, was introduced in [37]. Extending the same concept led to representations of even more complex constructs (e.g. functions or program submodules) as objects thus allowing greater abstraction [41, 27].

A well accepted method for representing source code is using Decomposition Hierarchies [42]. According to this framework all single entry- single exit programs can be represented as a structure consisting only of primitive program segments (sequence, conditionals, loops) also called normal forms. Using an equivalence mapping one can transform original source code to structured "code". Usually the source code is parsed and an AST is formed, then with consecutive tree to tree transformation we can obtain a tree in the form of a directed graph which will contain only normal forms.

Further use of dependency analysis tools can enhance each source code representation with the necessary adornments for further analysis. As a result the analyst will get several graphs showing:

- definition dependencies,
- calling dependencies,
- functional dependencies and
- data flow dependencies.

Combining the information from these analyses with one of the previously described methods the analyst can complete the first step of the design recovery process (see figure 2.1).

An experienced programmer can often reconstruct much of the hierarchy of a program's design by recognizing commonly used data structures or algorithms and knowing how they typically implement higher level abstractions. The higher the abstraction the easiest the understanding of the generic program structure [14, 58].

Britcher uses design languages to model programs as state machines (for data abstraction) and cartesian functions (for function abstraction) [7]. The resulting representation is translated in the design language providing the analyst with a pseudocode description of the source code. This approach is significantly different from other approaches that use condensed code listing because of its strong mathematical background and formality.

Presenting the user with a set of generalized control, data and call flow graphs is another approach [45]. The level of abstraction is usually controlled by the user. Each graph can be divided into prime subgraphs which have some basic functionality. Data flow diagrams and structure charts are used to model the data transformation aspect of a software system, since they deemphasize implementation details of the problem while focusing on the logical flow of data and control [28].

Smythe [61] replaces the intermediate representation with logical comments trying to start deriving the meaning of small pieces of code. The next step is the recognition of objects and object hierarchies, data are related to the procedures that operate upon them. In the last phase application domains are mapped to objects and constraints and system services to the user are identified (see figure 2.1).

Paul and Prakash proposed yet another approach in [51], they transform the original source code to a set of static relations describing code features (e.g. variables defines or used). Using this new intermediate representation the analyst can use all the commonly defined relational operators (e.g. joins, projections) or define new operators to aid in the analysis task he wants to perform.

Quilici [56] translates the original program into an Abstract Syntax Tree (AST) with frames which are used to represent each program action and its relationship to other actions. Actions are any units that the translator is capable of recognizing from language constructs.

ASTs are one of the most popular forms of intermediate program representation.

The creation of an AST is a three step process. Initially the program's source code is parsed using a grammar and a domain model created for the programming language of the subject system. While parsing the code structures, corresponding to the language basic constructs defined in its domain model, are created, populated and placed in a tree like formation. The final step in the creation of an AST is adding any additional information in the form of annotations in the nodes of the tree.

Rigi [46] uses entity relationship diagrams to represent static program relationships. A specific format known as Rigi Standard Format (RSF) is used to store these diagrams. The next step is to analyze the resulting RSF tuples in order to create visual images to facilitate program understanding and aid further analysis.

Abstract functional concepts can also be represented by programming plans or clichés. Possible components of a programming plan [22, 58, 19, 71] are the building components of an algorithm in terms of atomic program elements or other plans in the proper sequence (event path expression) [30]. Plan definitions are translated by a plan parser into inference rules as system's understanding knowledge. A pattern directed inference engine is then used for recognizing plans in a program and the whole understanding process is recorder by a Justification Truth Maintenance System. The effort here is the creation of a knowledge based system for program understanding. Several interesting issues arise by this approach, defining system's knowledge as plans, capturing all variations of an algorithm and guarantying completeness and correctness of the knowledge base are still major challenges.

Wills in [71] uses a graphical notation, called the *Plan Calculus* to facilitate understanding of complex annotated flow graphs that are used for plan description and recognition. This approach combines control and data flow graphs and is very descriptive but unfortunately not portable.

Hartman breaks down cliché recognition [31] to three major steps :

- a program representation or model,

- programming knowledge of standard plans, and
- search and comparison to find a plan instance.

The reader can easily see that this decomposition is equivalent to our design recovery process breakdown (figure 2.1). Cliché recognition is thus a significant step towards understanding the programmer's intentions.

All the above mentioned methods rely on the existence of an expert on the subject system for this second step. Everybody will accept that the easier solution to any problem is finding someone who knows the solution. Some claim that we are very far from a completely automated design recovery process [3]. A possible replacement of human experts is the existence of some knowledge base - domain model that could capture this necessary expertise. Biggerstaff [3] defines the domain model as "the knowledge base of expectations expressed as a pattern of program structures, problem domain structures, naming conventions and so forth, which provide a framework for the interpretation of the code". Building such a knowledge base is a non trivial task; it is the result of a process known as domain analysis during which information used in developing software systems is identified, captured, structured and organized for further use [54].

The main functionality of such a domain model is to include more information than the analyst can find in the code alone and thus guide and assist the code understanding process. Tools that respect the above mentioned guidelines exist and will be briefly presented in following sections.

The end of this second process step should leave the analyst with a library of recognized design abstractions. The next step is mapping the acquired knowledge to the source code (see figure 2.1). The underlying assumption here is that the analyst expects these abstractions to occur in multiple places in the code. Of course this is not guarantied, it is perfectly valid that the only occurrence of a concept will be on just one point in the code. Never the less one thing the analyst knows *a priori* is that his effort to locate the abstraction in the source code should at least yield one result if performed in the same piece of code that was used to "originate" the abstraction.

2.1.2 Concept to code mapping

The final goal of the design recovery process is to locate the occurrences of recognized abstractions in the source code. The task presents several challenges but certainly the most important one is the implementation of an algorithm to compare intermediate code representation and plan descriptions.

The ideal scenario would be to be able to deduce plan-source code functional-logical equivalence. This is an undecidable problem and in reality the most optimistic result any algorithm can claim is partial recognition. The expressiveness and the freedom provided to the user by currently used programming languages make recognition of equivalent plans a very difficult task. Problems related to concept-to-code matching are [58, 71]:

- syntactic variations of the same concept,
- parts of the concept might not be adjacent in the code - scattered concept,
- implementation variations,
- overlapping occurrences of a concept,
- unrecognizable code,
- variable aliasing and
- side effects.

Systems might also report incomplete together with multiple or unsuccessful recognition results. Using domain knowledge and information, besides the source code and the concept description, the analyst should be able to resolve ambiguities. If not, then incomplete bindings should be produced for further study.

Comparison-matching algorithms depend on the intermediate representation used to describe a concept. A quick look in the literature reveals a great diversity in the intermediate representations used in design recovery system, we will just mention a few.

The comparison algorithm in PROUST [35] matches syntax trees with syntax tree templates. In TALUS [48] user supplied function are compared with reference functions using heuristic similarity metrics. In CPU [40] comparison is done by applying unification and a matching algorithm on lambda calculus expressions.

Perhaps the closest approach to the one presented in this paper is the one used for PAT [30]; the original program is parsed and a set of independent objects (also called events) is created and stored in a repository called : the event base. These objects are subsequently used to recognize higher level events and function oriented concepts using a deductive inference engine.

In the Program Recognizer [58] a programming plan or concept is presented as a hierarchical graph structure composed of boxes which denote operations and tests, and arrows which represent control and data flow. Using this framework, plan (or cliché) recognition can be seen as a graph parsing problem which is the identification of subgraphs inside a larger graph that represents the whole program. When a cliché is recognized, its subgraph is substituted by a more abstract operation - node in the program graph thus forming an abstract and comprehensive image of the system.

For Quilici [56] programming concepts or plans are represented as data structures with two main parts: a plan definition, which lists the attributes of the plan that are filled in when instances of the plan are created, and a plan recognition rule, which lists the components of the plan and the constraints on those components. An instance of the plan is recognized in the AST, which serves as the program's intermediate representation, when all its components have been recognized without violating the constraints. The diversity is obvious, more systems are described later in the state of the art section.

An interesting issue is also the initial selection of possible candidates for comparison. In most systems the comparison occurs between the source code abstract representation and a plan expressed in the same abstraction formalism (graph, AST); in these cases a search algorithm is invoked to locate possible comparison starting points. Bottom-up approaches usually select all possible candidates found anywhere in the program's intermediate representation, while top-down approaches seek only specific parts that can satisfy a given subgoal.

If the program and the plans are not represented using the same formalism than hierarchical recognition control strategies are adopted. In this case complex plans are recognized in terms of their subcomponents.

To facilitate the comparison program, decomposition can be performed to produce program parts more likely to correspond to the plans. Program decomposition can be performed *a priori* before the selection starts or dynamically based on previous comparison results.

2.2 State of the practice

A variety of commercial tools capable of helping the analyst in his task of reverse engineering a system are available today. In this section we will describe some well known systems that focus on design recovery and program understanding. Most of these tools perform data and control flow analysis of the system. The ultimate tool for program understanding would include all the following features :

- a user friendly user interface,
- a local repository - knowledge base,
- several graphic editors,
- program fragment localization capabilities,

- redundant and duplicate code detection,
- dead code detection,
- powerful domain model browsing and editing,
- enhanced code browsing,
- simulation capabilities,
- on-line help and
- configuration and version management.

Using a combination of several available tools an analyst can use most of these features today.

The Software Refinery or simply Refine [39] is one of the most widely used tools in the reverse engineering field. The package consists of three tightly integrated modules :

1. a high level specification language,
2. an object oriented repository and
3. a language processing system.

There are also facilities for user interface extension. Refine currently supports four popular programming languages : COBOL, Ada, C and Fortran. The system takes the source code and parses it, using its language processing module. The result is an annotated AST which is stored in the tool's local workspace-repository. Several data and control flow analyses are offered and various reports can be generated (i.e. coding standards, variable and types reports). Using the specification language, which is a Lisp dialect, the analyst can perform further queries on the repository and implement algorithms to perform new analyses. The extensibility of the tool is one of its most compelling features.

VIA/Center is a Viasoft [12] product and focuses on COBOL systems. Offered analysis covers data structuring and relations as well as traditional control flow analysis. The results are stored in a specialized database.

Cadre technologies [12] offers a set of applications which are able to graphically represent abstraction hierarchies and also provide statistical information about program execution.

Design Recovery [8] is a product of Intersolv. The system can translate COBOL code to diagrams that clarify the underlying structure. To generate the physical models a local database of definitions is consulted and enhanced. The models can be examined, altered and then reused to generate new code. The tool has several other features like: dead code detection and complexity metric calculation for code segments.

LogiCASE [66] by Logic Technologies is a CASE tool that supports the maintenance and development of C programs and their corresponding detailed design. It can be used for reverse and forward software engineering and it offers design recovery from code as well as code generation from design. Design recovery tools transform selected code into a decision table. When the modification is complete, code is regenerated from design.

The TXL Transformation System [16] developed in Queen's University is used by Legasys Corporation for their products [17]. Legasys focuses on legacy code analysis and design recovery systems, with an emphasis on large-scale systems implemented in COBOL and C. The TXL Transformation System is presented in the next section.

FULCRUM 2000 is a product by Software AG [64], it is also an extension of the FULCRUM Workbench environment for long-term applications and design recovery.

At the Palo Alto Research Laboratories of Lockheed [44], a system called InVision is developed. It is used to renovate software, it was created to allow companies to modernize their legacy software assets, while incorporating contemporary data access standards, performance, and reduced maintenance costs. At the heart of InVision is a robust reverse engineering environment that uses object-oriented and expert system software design recovery technology.

Imagix produces Imagix4D [33] which is a program understanding tool. Imagix 4D, helps the analyst understand software that is complex, large, or unfamiliar. The tool provides modules for automatic exploration and documentation of code and use knowledge-based exploration and information visualization technologies.

Leverage Technologies [67] offers off-the-shelf tools for C, FORTRAN, Cobol, PL/I, and Ada based on the Software Refinery system. These tools can be used for: redocumenting and extracting design from legacy systems.

Several packages that allow smart code browsing have also been developed (Hypersoft [12] for COBOL and X technology [12] for C).

Other commercial systems (source [1]) are :

- Ensemble by Cadre ,
- Amdahl's Map Tool,
- Imagix- program understanding tools for C and C++,
- MOREIRA Consulting a tool for reengineering Legacy Systems,
- Strategix Reengineering Information Systems,
- Reading CASE Services, Reverse Engineering Tools,
- ASMFLOW by Quantasm Corporation,
- Bachman Re-engineering Product Set,
- Ernst and Young Redevelopment Engineering Tool Set,
- Intercycle by Interport Software Corporation,
- PACREVERSE,
- PATHVU by XA Systems Corporation,
- re/NuSys by Scandura Intelligent Systems,

- pSOSystem by MasterWorks,
- RXVP by General Research Corporation and the
- Sneed Tool Set.

Unfortunately detailed information about implementation issues for most of these systems is not publicly available.

All of the above systems although powerful can not carry through the whole task of reverse engineering a given system. Several attempts to create an integrated environment gain support and progress on domain analysis is probably the key to the problem. If a generic standard for an intermediate representation can be adopted by different tools then we will be much closer to the desired solution. Currently the analyst has to use several tools separately to achieve the results he aims for. Stepping to more experimental approaches we find a considerably larger number of systems.

2.3 State of the art

A multitude of significantly different approaches have been pursued focusing on the design recovery problem as part of the program understanding process. In this section we present some of the most well known systems that emerged from various research labs.

PROUST [35] can be viewed as an intelligent tutoring system for novice programming students. The target language is Pascal and the user should initially create a template describing the pattern he is looking for. PROUST uses a top-down control strategy applied to a solution goal tree. The matching occurs between templates and source code. Heuristics and a set of transformations are used for ordering, comparison, evaluation and search space minimization.

The TXL Transformation System [16] is a general purpose source-to-source structural transformation system. According to its developers, TXL can be used for source code analysis and migration, to program restructuring and design recovery tasks. Transformations are specified in the TXL programming language, a hybrid functional-rule based language with unification, implied iteration and deep pattern match. Each transformation specification has two components : a description of the structures to be transformed, specified as a grammar in unrestricted ambiguous context free BNF;and a set of structural transformation rules, specified by example using pattern-replacement pairs. TXL has been used to transform many popular programming languages.

Another system using knowledge-base tools for reverse engineering legacy systems is COGEN [43]. The system tries to capture and model the expert knowledge of software engineers in terms of conversion rules. COGEN uses an AST representation and stores it into a deductive relational database. The data definitions are captured in a symbol table. Queries can be entered into the database to obtain various kinds of useful information about the program's structure and behavior in terms of data and control flow analysis. To convert the program, the translation rules are applied to restructuring the program in the database, creating new facts describing the program in the new environment and altering the original syntax tree with new statements added and old statements commented out

Talus [9] is another system developed for intelligent tutoring. The target language here is LISP. The system is capable of automatic program debugging by correcting errors in LISP programs. To perform this task the source code is compared with correct code which has the same functionality. Comparison occurs between user supplied functions and reference functions from a library based on a heuristic similarity measure. To locate comparison candidates the system uses a A^* best first search algorithm.

Letovsky's system called CPU [40] represents programs as lambda calculus expressions and procedural plans. The system uses rewrite rules and a bottom-up control strategy. Top-level control selects and transforms lambda calculus sub-expressions applying all possible transformation rules until no more transformations are possible. Comparing candidate segments in CPU is done by applying a unification and matching algorithm on lambda calculus expressions.

A rule based approach is also followed in the Program Analysis Tool (PAT) implemented by Harandi [30]. The heart of the recognition system is a deductive inference engine. Initially an object oriented representation of the system is created after parsing the original source code. Rules are then used to describe plans and higher abstractions of objects and function oriented concepts.

Object oriented representations of code are also used in a number of systems [27, 41, 18]. The SAMS system [37], for example is actually implemented on top of an object oriented DBMS.

Systems that use an AST intermediate representation are the RECORDER [10] and PECAN [57]. PECAN is a smart code browsing system. Source code is parsed and an AST is created the source may be viewed in a number of different ways. The code itself may be pretty-printed with multiple fonts, as a structured flowchart, or as a module interconnection diagram.

Using graphs as the main representation formalism led several researchers to develop systems that are actually comparing graphs. The following six systems fall in this category.

In UNPROG [32], the abstractions used have the form of control and data flow graphs. The user specifies a programming plan in the same terms and then the source code control and data flow relations are compared with the programming plan's control and data flow graph relations. If we can prove that a subset relation exists then the user specified plan is recognized.

Quilici's system [56] tries to match structurally frame schema representations of C code. If the match is successful then data flow graphs are compared. Candidate plans are selected based on an indexing scheme. After a successful match semantic abstractions occur by substituting the selected frame with the abstracted one. The process continues until no further abstractions can be generated.

In [20] a design recovery prototype is described. The system works on a subset of Modula 2 and uses graphs. The original code is parsed into an intermediate form called Program Analysis Graph (PAG). Further analysis of the PAG with the aid of a knowledge base leads to a transformation into another more abstract PAG. Finally, translation of this resulting abstract PAG into the user required form occurs. This form can be a program in the original or in another programming language, or even readable documentation.

Influential work on graph parsing is done also in the Programmer's Apprentice Project [58], the Program Recognizer [70] and their successor GRASP [71]. Attributed graphs are used to represent programs and thus subgraphs represent programming plans. The system performs bottom-up graph parsing using a context-free graph grammar representing standard transformations between standard plans and semantic abstractions for already recognized plan instances. Parsing checks all possible subgraphs thus all possible interpretations can be found and be represented in a lattice. The actual comparison is performed by matching subgraphs and by checking constraints involving control dependencies and other program attributes. All three last mentioned system depend on analysis of the low-level formal details and therefore emphasize a full and exact match for recognition. The computational load required suggests that scaling up to industrial sizes will be quite difficult.

The work by Arango [23] has solved the scaling up problem but can't create abstractions as generic as other systems (see Desire). Arango's system (Draco) focuses on the structure of the transformations and the operations on transformations trying to completely automate the recovery process. To achieve this all informal information is completely ignored.

Desire [3] works on C code and implements several of the ideas presented in the theoretical part of this chapter. In this system C code is parsed and several parse trees are produced. A set of postprocessors use these parse trees and a dictionary containing higher level information about functions, files and global data is produced. The next step is the creation of a plane-text web by postprocessing the abstractions. The analyst can then write Prolog statements in order to extract information from the stored abstractions.

The SCRUPLE [51](Source Code Retrieval Using Pattern LanguagEs) system developed in the university of Mitchigan is based on a pattern query language. The analyst uses this language to specify structural patterns of code. The degree of precision can be adjusted be using different language mechanisms. The user specified pattern is checked against the parsed source code which has the form of an AST. To allow users to express more powerful queries a source code algebra is defined. Queries can thus be optimized using algebraic transformations rules and heuristics.

However powerful analyses all these systems can perform none can claim efficiently solving the main problem which is design recovery. Corbi states that automatically recapturing a design from source code is not considered feasible task yet [15]. The obvious question now is how can we get the most out of the existing tools. The answer is integration.

Tool integration and increased interoperability of tools represent major current trends. This is evident from the extensive efforts toward improved integration between front-end tools and code level tools. Integration will enable more adequate support for both forward and reverse engineering[60]. The next section describes our experience trying to build an integrated environment and how it relates to the work described in this report.

2.4 The REVENGE project

Manny Lehman observes that any software must continually change or become less useful in the real world. This was exactly the problem with the Structured Query Language/Data System or simply SQL/DS. SQL/DS is a large relational database management system that has evolved since 1976. Based on a research prototype after numerous revisions it was first released by IBM in 1982. The system was originally written in PL/AS and then migrated to PL/X. PL/AS is an IBM proprietary system programming language. The system now consists of more than three Million Lines Of Code (MLOC). The target of the REVENGE project was to use several complementary reverse engineering technologies on this real world system to help its evolution and maintenance.

During evolution inevitably the structure of a software system will degrade unless remedial action is regularly taken. The problem is that for most legacy systems no remedial action is ever taken and as a result the system after several evolution cycles becomes completely unstructured [2].

Some of the initial goals of the project were:

- detecting uninitialized data, pointer errors and memory leaks,
- detecting data type mismatches,
- finding incomplete uses of record fields,
- finding similar code fragments,
- localizing algorithmic plans,
- recognizing inefficient or high complexity code,
- predicting the impact of change and
- creating a framework for the integration of the resulting systems.

The main constraints were ensuring code correctness and performance enhancement.

System components

To achieve the given goals six systems were selected or developed by different teams.

1. SCRUPLE from university of Mitchigan,
2. Rigi from university of Victoria,
3. Ariadne from McGill university,
4. Telos from university of Toronto,
5. a filtering detection system from IBM Toronto Labs and
6. a text redundancy recognition system from NRC.

All tools were tested using C programs as subject systems but should also be able to handle PL/AS code with little or no modification.

The IBM system [11] performs defect filtering using the commercial product Software Refinery.

The NRC system [34] identifies the exact repetition of text in huge source code. The approach works by fingerprinting an appropriate subset of substrings in the source text. A fingerprint is a shorter form of the original substring and leads to more efficient comparisons and faster redundancy searches.

The three first systems focus on pattern matching approaches of the subject system in different levels. SCRUPLE was described in a previous section.

Rigi [46] was used to assist the system's redocumentation. The source code is parsed and the resulting artifacts are stored in a local repository. Using these artifacts we can create a flat flow-resource graph of the system. This first fully automated phase is followed by a semiautomated phase in which the analyst explores interactively the system using his/her pattern recognition skills and language-independent subsystem composition techniques provided by Rigi. The result is the creation of subsystem hierarchies. A multitude of views of these hierarchies can then be created.

Evaluation and understanding of these views can aid efficient redocumentation of the subject system.

The repository developed in the university of Toronto is called Telos [24]. The group in Toronto was in charge of developing an information schema - domain model that could be "understood" and used by all the tools involved in the project. The repository using this schema should be able to save all the artifacts of the various analyses performed by the cooperating tools. To minimize the workload for this global repository each tool only stores in it, data required by other tools. The rest of the analysis information resides in the each tool's local workspace and can be sent to the repository if requested.

Ariadne [38, 25] tries to address three important problems:

1. produce intermediate representations able to capture structural and semantic aspects of the system,
2. automatically locate similar fragments of code (code cloning detection) and
3. partial recognition of programming plans or intents in the source code.

As we saw in previous sections a variety of intermediate representations exists. Ariadne uses an object oriented annotated AST. The AST is created after source code parsing using the Software Refinery's language processing module enhanced with our domain model and grammar for the C language. The resulting AST is annotated with important information computed by several data and control flow analyses. Every node in the AST is adorned, among other information, with :

- source code location,
- links between identifier references and corresponding variable and data definitions,
- variables used and set,
- functions called,

- variable scope information.
- input/output operations.
- a series of complexity and quality metrics (D-Complexity, fan-out, McCabe, Henry-Kafura's information flow quality and Albrecht's function point quality metric)

In large legacy systems code duplication is a common problem. Programmers trying to extend the system's functionality tend to "cut and paste" pieces of code in order to reuse it somewhere else in the system. As a result code modularity is destroyed and existing bugs in the initial code are replicated. If the code remains unchanged then the NRC tool can trace it but if even slight changes are made, the fingerprint approach is no longer effective. The task of comparing functionality of two code fragments is still an open theoretical issue. However applying heuristic rules can provide us with an initial answer which the analyst is subsequently called to validate. The assumption we made for our heuristics is that similar pieces of code have similar feature and metric values.

To implement our solution [26] for the second task (localization of similar code fragments) the annotations in the enhanced AST were used. The metrics used as heuristics are:

1. fan-out which is the number of functions called from a source segment,
2. the ratio of input - output variables to the fan out,
3. McCabe's cyclomatic complexity,
4. Albrecht's Function Point quality metric and
5. Henry-Kafura's information flow quality metric.

Comparisons are made using the Euclidean distance defined in the five-dimensional metric space and clustering thresholds defined on each individual measure axis. Further grouping of code segments based on criteria such as shared data references and data bindings is also performed.

The final task was plan localization, the solution implemented in this project was the inspiration for the work reported in this document. As we saw earlier graph based solutions to this problem result in computationally expensive and complex algorithms. On the other hand algorithms using plain textual-lexical matching fail when plans are delocalized or contain "noise" in the form of irrelevant statements. Also algorithms in the last category cannot possibly capture any behavioral information about the system.

We believe that a fully automatic approach based on an incorporated library is not fit for our task. Having to reengineer proprietary code one does not have the luxury of access to a vast collection of plans in this language. Our algorithm encourages human assistance. Plans have the form of portions of the annotated AST and are expressed in a rather powerful language we call Abstract Concept Language (ACL). More details about our approach will be given in a following section.

Figure 2.2 shows a high level module decomposition of Ariadne. Main system activities are depicted as separate modules, each module is described briefly in the following paragraphs.

A typical session using Ariadne would be the following: the user chooses the piece of C code he is interested in analyzing and then parses it using the built-in parsing facilities of Refine in order to create an object-oriented AST which will be used for further analysis. Refine provides a standard domain model for the C language which is extensible and can be augmented to include any additional information the analyst deems necessary.

The first step after the creation of the AST is the calculation of a series of metrics which is done by the Metrics Calculation Module. Metrics are used in almost all further analysis. For example the user can identify similar code fragments (also known as clones), this is possible by comparing metric distances (absolute or euclidean) of candidate code fragments. Using metrics which are actual real numbers instead of vectors of features simplifies and accelerates the whole process: code cloning detection functions are part of Code Cloning Detection Module. Several dataflow analyses of the target piece of code are also possible (i.e. common references, data bindings) as parts of the Dataflow Analyses Module.

Another separate module is the one that constitutes the prototype based on which we developed our system. The Programming Plan Recognition Module focuses on identifying code abstractions described in an Abstract Concept Language (ACL) in C programs. This module is based on the theoretical background described in chapter four.

Finally Ariadne has the ability of storing analyses results (and any other object in its object-oriented AST) in a centralised object-oriented repository that can be accessed by other cooperating tools. Communication with the repository is possible through two modules that handle the downloading and uploading of the AST as well as other synchronization issues.

Implementing a way of integrating the various involved tools was a core requirement of the project. In CASCON'95, a conference organized by IBM's Center for Advanced Studies laboratory in Toronto, we demonstrated the final product and showed the implemented capabilities. In the next few paragraphs we will try to present the environment's architecture and analyze how we implemented two way tool communication.

Making tool interaction possible

Integrating different reverse engineering tools to supply the analyst with enhanced functionality is a major trend in the field. The key issue, in this effort to create such an environment, is the adoption of some common source code representation to serve as a communication standard. For us this standard was the global schema used by the repository. The basic requirements for the global schema are :

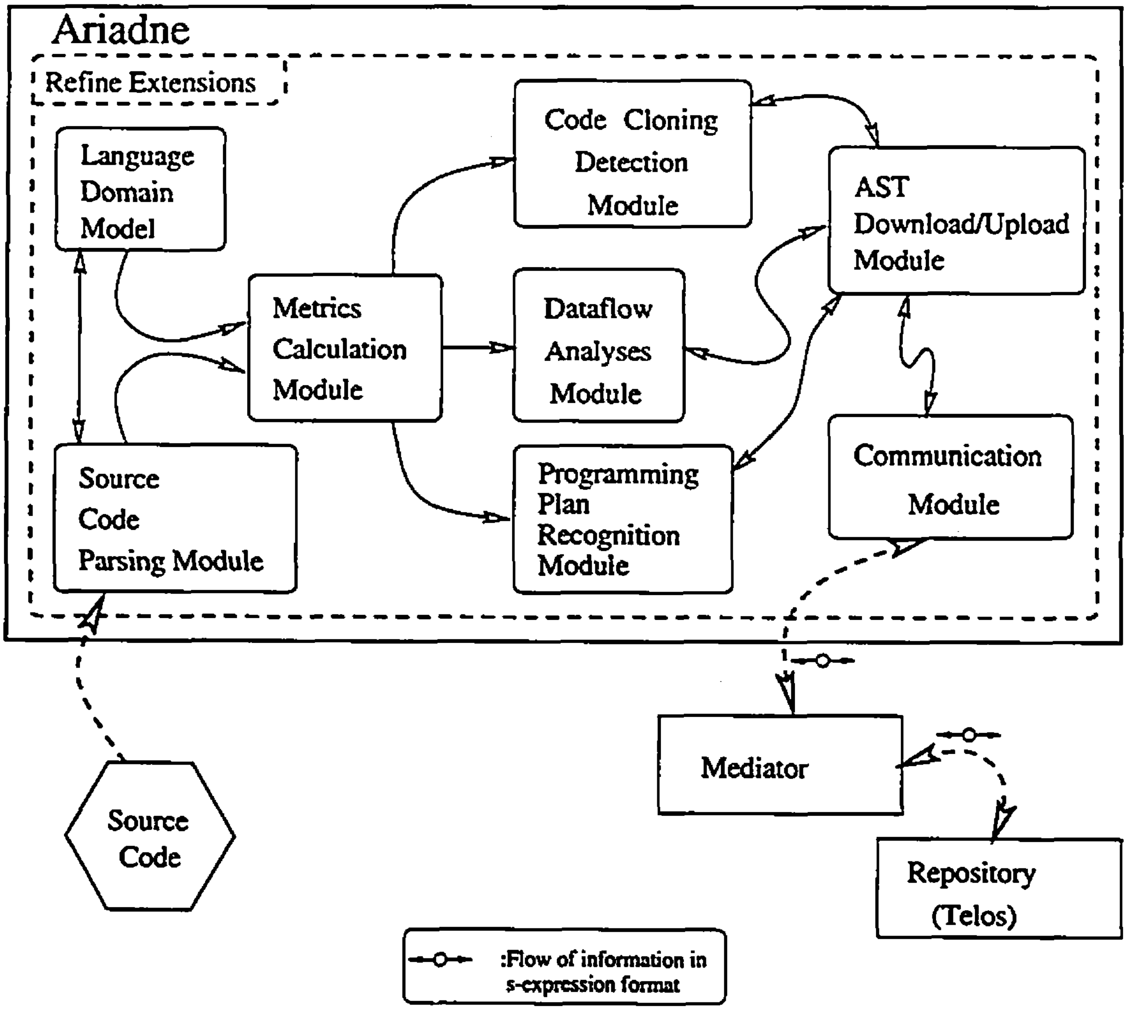

Ariadne
Refine Extensions
Language Domain Model
Code Cloning Detection Module
AST Download/Upload Module
Metrics Calculation Module
Dataflow Analyses Module
Source Code Parsing Module
Programming Plan Recognition Module
Communication Module
Mediator
Source Code
Repository (Telos)
:Flow of information in s-expression format

Figure 2.2: Ariadne's module decomposition.

- completeness and
- flexibility - extensibility.

The schema should be able to capture artifacts from different cooperating tools. The main strategy to achieve these goals was creating object metaclasses and classes for all common objects in different tools as well as specialized classes for tool dependent objects. For example both Rigi and Ariadne can have the notion of a function and thus the creation of one class with attributes that can capture all possible information generated by each tool was the solution. To capture objects particular to one tool in the environment, tool-specific subschemas were designed and implemented.

The next phase was detecting possibilities of tool cooperation. Each tool's functionality can be complemented by some other tool's capabilities thus leading to new analysis possibilities and generating novel views of the subject system.

Telos being an object oriented repository provided an excellent platform for the resulting schema. Having achieved data integration using the schema we had to ensure control integration. Control integration was made possible through a customizable and extensible message server named Telos Message Bus (TMB).

In order to send an object's description to the repository the s-expression formalism was used. As we already mentioned the repository's global schema describes all possible object classes. When an instance of a class has to be stored its attribute values are sent to the repository. An instance of a program with only two attributes (the program directory location and name) described in s-expression format would be :

```
(Program_1242 Token
   (Program)
   () (
    ((programDirectory)
     (("/reverse/data/src/list")))
    ((programName)
     (("list")))))
```

Analysis of this s-expression reveals the following points : firstly an identification string for the object (Program_1242) is given then the object's tier is specified. Telos allows three possible object ranks :

1. Metaclass : objects of this rank are used as class generators,
2. Class : objects in this tier are actual class definitions,
3. Token : token objects are instantiations of a class.

Having metaclass and class tiers allows each tool to dynamically expand the schema by sending a new metaclass or class specification always in the form of s-expressions. An example of such an s-expression follows.

- **Metaclass**

```
(RefineClass M1Class
  ()
  (ObjectClass)
  (((attribute)
    ((refineNonTreeAttribute Proposition)
     (refineTreeAttribute Proposition)))))
```

- **Class**

```
(ExtractionObject SClass
  (ObjectClass)
  (Object)
  (((attribute setValue)
    ((allRelevantObjectsToAnalysis Object)))
   ((attribute singleValue)
    ((correspondingCode ProgrammingObject)
     (analysisName String)
     (dateOfAnalysis String)))))
```

Secondly the s-expression description for the program token references the base class of the token (Program). Thirdly the pair of empty parenthesis that follows is reserved for the token's ISA class. In our case is the same as the base class and thus omitted. Finally following these necessary basic fields, the names and the corresponding values of each attribute for the object are sent. Attribute values are classified in the following categories:

- single value attribute (String,proposition),
- set value attributes (SetValue) and
- sequence value attributes (default).

Another example of an s-expression follows, here the reader can see the values passed for some of the metrics and attributes that we use for our analysis.

```
(Function_1243 Token
   (Function)
   () (
    ((albrecht)
     ((23.0)))
    ((dComplexity)
     (( 1.5)))
    ((fanOut)
     (( 1.0)))
    ((functionDefBody)
     ((Block_1244)))
    ((functionDefParameters)
     ((DeclarationSubtree_1245)))
    ((functionName)
     (("elementcreate")))
    ((identifiersUsedNames)
     (("i")
      ("_iob")))
    ((kafura)
     ((576.0)))
    ((location)
     (("element.c:13,25")))
    ((mccabe)
     (( 2.0)))
    ((variablesSetInConstrNames)
     (("info")
      ("next")))))
```

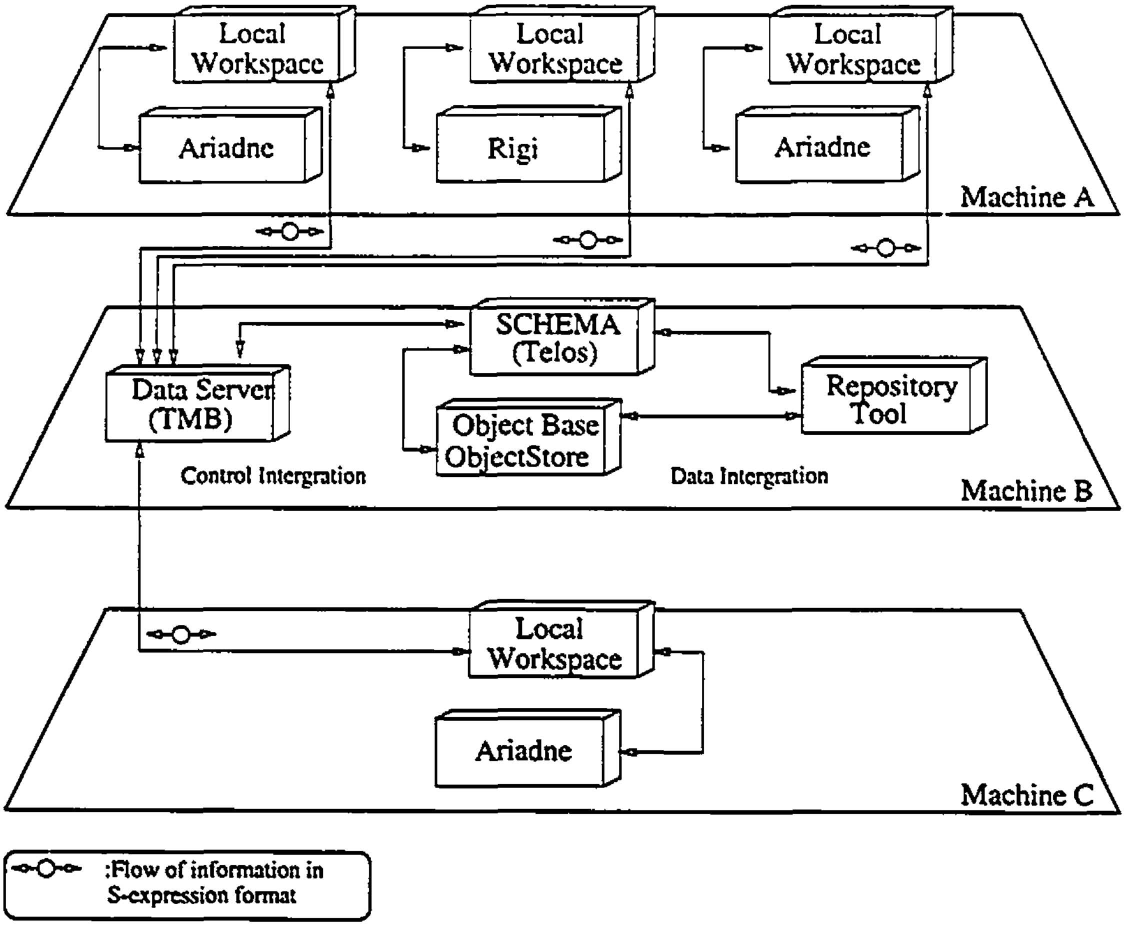


Figure 2.3: The system's architecture.

In an implemented scenario of meaningful tool interaction Ariadne produces analysis objects which are sent to the repository. Rigi downloads these objects, uses them to perform analyses not supported by Ariadne and then uploads the objects enhanced with the new analysis information back to the repository. Ariadne can then retrieve these objects and perform additional analysis. A delicate issue here was mapping the retrieved objects to objects back in Ariadne's local workspace. The implementation of a mechanism to accomplish this task and ensure atomicity between the transferred objects, the evolution of the user interface and the communication module for the Ariadne system were the writer's contribution to the REVENGE project.

The overall system's architecture is shown in figure 2.3. Various CASE tools (i.e. Ariadne, Rigi) are running in different machines across the network performing analysis on the same or different subject systems. Resulting information is passed to the *Data Server* in s-expression format, stored in the knowledge base and sent upon request to any cooperating tool.

2.4.1 The influence of REVENGE

Our involvement with the REVENGE project had a major influence on the work described in this report. The decisions we took based on our experience building and using REVENGE are :

- adopting the algorithm for code segment localization previously implemented for Ariadne in Refine,
- making the new tool part of the REVENGE environment.
- using parts of the domain model for the global schema created for REVENGE,
- keeping the s-expression formalism for our communication with other tools in the environment.

Studying the algorithm used for partial recognition of programming plans or intents in the source code we felt that a more generic version of the algorithm could be used to achieve code segment localization in different programming languages.

The key idea was to keep the essence of algorithm but change the structures upon which it operated. The methodology is thus the same but the overall design is different. We still represent nodes in the AST as objects but the design of the class hierarchy and the way the algorithm is implemented and distributed among the classes make the new system generic enough to be used with different languages.

The approach for code segment localization resembles the one described in SCRUPLE [51] allowing for a similarity score to be computed between a query and a retrieved component, but offers significant enhancements in the query language and the comparison method. The complete algorithm will be presented in the next chapter in the design section.

Our new tool can be part of the integrated reverse engineering environment we described. Being compatible with REVENGE means being able to receive our input and send our output to other tools which respect the global schema. As we will show in the next chapter this fact presented several advantages.

We want to make clear at this point that the work described in this document is not merely "porting" the algorithm implemented in Ariadne to a new software platform. The new system presents a major difference: it is based on new, flexible and extensible framework and consequently its implementation is far more generic than the one in Refine. To place our system in the general design recovery process shown in figure 2.1 we can say that it focuses on the last step of the process which is mapping abstractions to the source code. The following chapter will make all these statements more clear to the reader by documenting the whole process of building the system.

Chapter 3

Gathering System Requirements

This chapter discusses the first steps toward the creation of a generic framework which will be used to implement a new system for code segment localization. The new system is based on the algorithm used in the prototype built for the REVENGE project. Motivation for building a new system will also be discussed. The purpose of this work was to extend and generalize the prototype's functionality and domain. In the following sections we explain in detail the process of capturing the core requirements for this new system.

3.1 Adoption of macro process

One of the first requirements for the new system was to implement it in a platform-independent, popular, object-oriented language. Having chosen C++ as the implementation language we tried to find in the literature an appropriate framework that would help us formalize the development process. The process adopted was the one proposed by Booch [6]. In the next sections we will describe our actions to accomplish each step of the process. The macro development process consists of five major activities (see figure 3.1) :

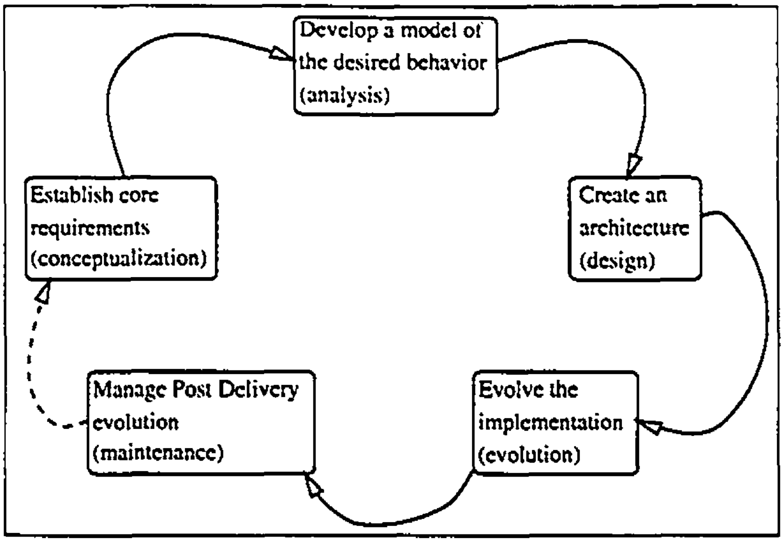

Develop a model of the desired behavior (analysis)
Establish core requirements (conceptualization)
Create an architecture (design)
Manage Post Delivery evolution (maintenance)
Evolve the implementation (evolution)

Figure 3.1: The macro development process.

1. Establish the core requirements for the software (conceptualization)
2. Develop a model of the system's desired behavior (analysis)
3. Create an architecture for the implementation (design)
4. Evolve the implementation through successive refinement (evolution)
5. Manage postdelivery evolution (maintenance)

Although the formal definition of the macro process may seem trivial to every experienced developer we found it particularly useful as a mean of structuring this chapter in a coherent way. In the lifetime of our system we had the chance to perform all the five activities mentioned and we are repeating the process trying to maintain the system. Adding new features and porting the system to other platforms are the activities currently performed.

3.2 Conceptualization

The main purpose of this activity is capturing the core requirements of the system. As we mentioned earlier a functional prototype of our system was already developed for the REVENGE project in a completely different implementation language (Refine). The existence of this functional prototype made conceptualization significantly easier, we no longer needed to spend time trying to prove that our algorithm can deliver results. The main objective was to prove that the algorithm can be improved by using a whole new framework and design in a different implementation language. Based on these ideas we captured the major functional requirements for a system using this new framework, the new system should:

- have at least the core functionality of the prototype system,
- be developed in such a way so it would be able to accept, as input, code from various "programming" languages,
- be compliant with the main architectural concepts of REVENGE so it can be part of this larger cooperative environment,
- add new features and explore other possible improvements,
- be implemented in a commonly used object-oriented language,
- conform with various standards of object orientation (design and implementation standards),
- be portable in all major hardware platforms.

Let us briefly analyze these core requirements and explain their rationale.

Duplicating the main functionality of the prototype system

Functional compatibility with the prototype system was our primary objective, we decided that in order to be able to evaluate our work a working system that could be tested against our prototype should be developed.

One main challenge was to maintain the efficiency of the algorithm in this new implementation. Refine has a variety of built-in, optimized functions to manipulate the AST that it creates. The algorithm for code segment localization is not very complex, the most critical functions are actually those that traverse several different structures and perform element retrievals or comparisons. Obviously the most difficult part would be the design of new structures and the implementation of algorithms for their manipulation.

What exactly we mean when we refer to the main functionality of our prototype is the ability to localize segments of "code" based on an abstract description of these segments.

Accepting different kinds of input-"source code"

Our prototype proved the capabilities of the algorithm, the idea that stimulated this research however was that the same algorithm based on a more generic framework would be able to perform similar tasks with a variety of inputs. The initial input is code in some "programming" language (C, Pascal or even HTML). The only constraint is the existence of some kind of structure in the language so it would be feasible to create a meaningful intermediate representation fit to use with the algorithm. When we refer from now on to "source code" we mean any possible structured input and not only the artifact of a specific programming language. Thus the terms input and "source code" are interchangeable.

The rising issue here is to find a formal way of representing the input, capable of capturing all our target domains (languages). The use of various intermediate representations is common practice in all reverse engineering systems that perform design recovery as we saw in the previous chapter. The basic advantage of any intermediate representation is the ability to capture only the aspects of the "source code" that are significant to the analysis performed while ignoring any other elements that may slow or clutter the analysis. Unfortunately there is no consensus on an intermediate representation but there is a wide adoption of ASTs (Abstract Syntax Trees) as a form of intermediate "source code" representation.

ASTs represent "code" in a structured way allowing on the same time annotations. Thus users can adorn each code element, represented as a node in the AST, with the attributes they deem necessary for their analysis. The s-expressions formalism was used for describing the building blocks of our AST. The decision to use s-expressions was unavoidable because of the next core requirement.

Compatibility with REVENGE

REVENGE, as we already described in the previous chapter, is a powerful environment for cooperative reverse engineering. We share the common strong belief among many researchers in the reverse engineering field [60, 69, 55] that in the future the ability of any CASE tool to cooperate with other tools as a part of a larger integrated environment will be a critical factor for its success.

Our experience building and using REVENGE proved that such cooperation is feasible. Conformity with a global schema and adoption of formalisms for the exchange of data between tools in the environment was the solution proposed in the REVENGE project. The experiences we acquired from our involvement in this project led us to choose the formalism to create our intermediate representation and also guided us to several important decisions about the system design.

To create the object oriented AST, which will serve as our intermediate "source code" representation, we had to have a parser for our input. It was clear to us that the main focus of this research is not to build parsers for all possible target languages (i.e. C, Pascal or HTML). Other tools in the REVENGE environment, namely Ariadne, provide specialized modules to accomplish this task.

Using Ariadne for the parsing permitted us to focus on our main research topic, the design of a generic framework. To create the AST we need to get from Ariadne directly, or indirectly from the repository (Telos), a description of each node using the standard formalism in REVENGE (s-expressions) and then reconstruct an image of Ariadne's AST. As we mentioned earlier the communication module used to send and receive information from the global repository as well as the facility to dump Ariadne's AST in s-expression format already existed and were parts of the writer's work for the REVENGE project.

Extending our prototype

In addition to the conception of a system architecture that can handle several different "code sources" we tried to explore other possibilities for our system such as ways of improving functionality, flexibility and user friendliness.

All systems that perform concept recognition depend on some sort of feature comparison, ours is not an exception to this rule. However the ability of adding new features or changing the feature comparison method is not usually supported by most systems mainly because of their rigid design. The design of our system allows such changes by incorporation of add-on (plug and play) modules. Creating these modules is a low effort programming task.

Another frustrating issue for end-users is usually the learning curve necessary for a productive usage of the system. In most systems performing design recovery, new language or formalism is introduced to describe patterns. This is of course a necessity and can't be avoided in systems that need to have some sort of plan-concept description. Learning to use all these query-languages in an effective way can be a time consuming task. To improve user friendliness and ease-of-use a powerful and intuitive user interface was built to be part of our system.

Implementation language, portability, conformity with standards

The design of our cooperative reverse engineering environment was based on a common domain model which was shared between all tools and also served as the schema for the centralized repository. A substantial amount of work was devoted to the effort of creating this domain model and the result was an extensible design of several metaclasses and classes that could be used.

We spent time going through this design again and we felt confident that the new system in order not only to be compliant with REVENGE but also with current trends in software development should be implemented in a popular and powerful object oriented language. We chose C++ mainly because of our previous experience with it.

Another concern for us was the development process itself. We considered a great opportunity to put in action new methodologies for object oriented development. We decided to adopt a general framework for our process and adhere to coding standards so we can ensure extensibility and maintainability of our system.

Portability was another related issue, one of our main concerns for the success of our prototype was that being based on a commercial and not quite wide accepted yet implementation platform (Refine) it would be really hard to evolve and maintain. An implementation of the system using an object oriented programming language like C++ would help us to overcome these problems.

The above mentioned requirements are also the major constraints and measures of success for our system. In the next sections we will describe how we attacked the problem trying to satisfy all these core system requirements.

3.3 Analysis

Among several methods to facilitate system analysis proposed in the literature, we chose to adopt the use-case analysis method introduced by Jacobson because of its intuitiveness and effectiveness.

According to the use-case analysis method, all affected project members come up with possible scenarios fundamental to the system's operation. These scenarios collectively describe the system functions. Analysis then proceeds by a study of these scenarios to : identify primary function points of the system, group function points into clusters of functionally related behaviors and generalize primitive functions to create higher level abstractions.

The following section presents some possible scenarios for the system, mainly on the design recovery realm.

3.3.1 A view of the problem

The purpose of the following paragraphs is to present possible cases where our system could be used to handle problems which can't be easily solved using existing tools. Let us examine a few possible scenarios.

First scenario: Identifying error prone code

In legacy systems when a part of code is identified as error prone usually maintainers try to discover similar or identical code in other modules of the system. The problem that arises in this case is that the identified code might be slightly altered in other modules. Variable names might be changed, comments added or even the sequence of commands altered.

Second scenario: Identifying common source code patterns

It is often the case that the legacy system we need to reengineer is based on a proprietary language. Usually in this case the maintainer has access to other forms of code representation and secondary information about the system (i.e. control and data flow diagrams). The task is to understand the system module by module using accumulated knowledge of the system. Tools to aid engineers in their task using only secondary information rely heavily on identifying common source code patterns between modules.

Third scenario: Training

While learning a new programming language students learn to categorize language commands based on their functionality, they learn for example that a while statement is a special case of an iterative statement. High level algorithms are consequently introduced and the students are asked to implement them. Following this logic it would be beneficial for the student to have a tool able to recognize pieces of code that can be described by a certain abstract code pattern.

Fourth scenario: Software migration

In the process of changing the design of a system from procedural to object oriented maintainers need to identify key data structures and functions that manipulate these structures. Performing this kind of exhaustive searches in a multi-million line legacy system is surely not a trivial task. If the analyst can come up with the necessary information (i.e. data structure definition and key functions using this structure) then he can explore possibilities for code parameterization and class creation.

3.3.2 Use-case analysis

People in all the above scenarios share a common problem in different levels. We will attempt to analyze these scenarios to detect common entities and abstractions, this is a common approach followed for the creation of frameworks. For this task we adopted the process suggested by Schmid in [53], according to this method systematic construction of frameworks can be broken down to the following steps:

1. perform domain analysis with an aim to identify the fixed aspects that are common to all applications from the domain - called the *frozen spots*, and the

variable aspects, in which different applications may differ - called the *hot spots* of the framework,

2. derive a specialized model by an object-oriented analysis from a specific application or configuration of the domain,
3. generalize this model by a sequence of transformation steps one per hot spot.

The next paragraphs identify possible frozen and hot spots in the scenarios.

In all the scenarios we have an initial source of information, but with some important differences. In the first case our input is source code from a legacy system, the maintainer is probably familiar with the language and if he is lucky the source code is also well documented. We can say that it is a typical case of "rich" input which suggests a lot of capabilities for analysis. The second case is different, the input is in a proprietary language or may be in an intermediate representation of this language. The analyst is not probably very familiar with neither, but he has access to a domain expert and several analysis tools. In the third scenario the "analyst" is not familiar with the language at all and is actually going through a learning process. Lastly in the fourth scenario the analyst is quite familiar with the source code language and the functionality of the system.

The input or "source code" form is not the only interesting element in these scenarios, let us observe the desired result. In the first and fouth case the analyst has identified the part of the code that interests him/her and just wants to find all possible occurrences of functionally equivalent code. In the second case the analyst has probably recognized a few critical parts of the code, each one has a discrete functionality and combinations of them implement a larger logical task. The required task in these cases is the localization of these code segments. In the third case the "analyst" has for a informal description of a logical concept with which he could localize and observe actual implementations of this task.

Finally the missing link in all the scenarios is of course the system that could deliver the desired results. We will try to summarize our observations from these scenarios by answering some simple questions :

1. What might be the input of the system : Any structured "source code", either complete logically and physically or even incomplete or partial (frozen spot).

2. What is the primitive tasks the system should perform : The basic functionality is source code segmentation and localization of an abstractly described code segment in the code. Combining code segments will solve the more complex cases of concept localization (frozen spot).

3. How does the user describe a segment : This is on purpose the only issue not mentioned explicitly in the scenarios presented. As we can see in the first scenario the user has the actual statements in front of him and can use them as guidelines to describe what he actually is looking for. In the second scenario the analyst has only a partial description of what he wants . This partial description most probably will focus on specific properties that the segments or tasks should have ignoring small implementation details. In the third case assuming a given example in natural language or pseudocode the "analyst" should come up with a generic description of the task. The level of familiarity with the language used in the programs also varies.

4. In what form are the results presented to the analyst : Since detection of logically equivalent code is not possible with absolute certainty, the analyst is presented with a similarity measure indicating the system's belief that the abstractly described code segment is logically equivalent with the reported source code segment. The calculation of this measure is based on feature comparison between the two pieces of code (query and actual source code). Partial plan recognition is also possible and acceptable.

It is obvious that the analyst should have the ability to describe a segment either in extreme detail or in various degrees of abstraction. A way to achieve this is to provide the analyst with an Abstract Concept Language (ACL) [26] to describe the code segments. The ACL language should use keywords similar to the target (input) language so that it would be easier for the analyst to describe a concept just by looking at an instance of it in the source code that implements it. Thus the code description varies depending on the target language and can be characterized as a hot spot.

As Booch notices [6][p.252] analysis is impossible to be completed before design commences. With the information we have at this point we can form a first generic design of our system.

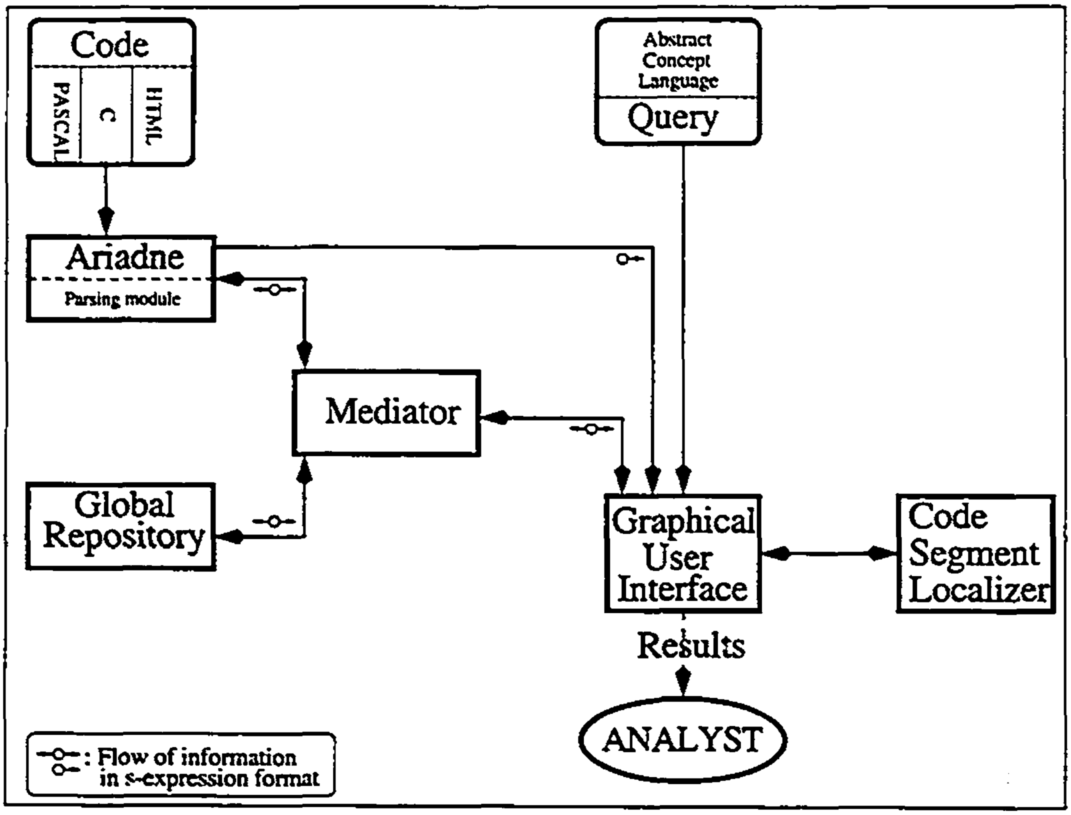


Figure 3.2: General view of the system.

We can divide the system into two major components. The first component is responsible for supplying the system with the necessary information to build the

AST and an abstract description of the code segment we want to localize. The second component consists of the main program that implements our algorithm (namely the Code Segment Localizer or simply CSL) and a graphical user interface (GUI).

3.3.3 Hardware and software requirements

The system was implemented in an IBM RS/6000 workstation using the AIX operating system. As the reader can see in figure 3.2 the system uses several tools. The vital part of the system however is the CSL module and the GUI. Both these modules are developed using languages which are portable to all commonly used platforms. CSL is implemented in C++ and the GUI in Tcl/Tk.

Tcl/Tk was chosen as the GUI development language for two reasons: its portability and most importantly our prior experience using it in various projects. We found Tcl/Tk to be an excellent rapid application development tool, using several library extensions of Tcl/Tk we built a robust and intuitive GUI to facilitate interaction with the system.

The CSL module uses Lex and Yacc for the parsing of the input (s-expressions describing the AST and the query describing the code segment we are looking for). All the above mentioned third party programs are implemented for various platforms and our modules do not have any specific hardware requirements. As a result we can claim that our system is platform independent.

3.3.4 Analysis conclusions

In order to test the framework we built a system which can be used to assist the analyst in the design recovery process and the concept assignment problem [4]. In other words the system assigns a physical location, in the source code, to an abstractly described concept in a query. The process of recognizing large-grain, composite concepts or plans requires that we first recognize the elemental concepts which form the larger concept. The system will have the ability to recognize this fine-grained concepts and then, using an inclusion mechanism, put them together to form and subsequently recognize larger-grained concepts (*hierarchical recognition*).

The primitive operation to complete the task we just described is *code segment localization*. The analyst supplies an abstract description of one or several code segments expressed in a language with the same low level representation as our intermediate representation of the initial input-"source code". We reconstruct the AST, which is the intermediate representation of our "source code", given the s-expression description of its initial nodes either from the global repository through our mediator module or directly from Ariadne. The CSL module then tries to locate the specified segment abstractions in the AST and reports successful attempts to the analyst using the GUI module. Each result reported provides the analyst with the exact location of the code segment in the "source code" and a probability indicating our belief that the given abstract description matches the code reported. To calculate the result our matching algorithm compares the formal, structural features of the code segment pattern described by the analyst with parts and their corresponding features of the reconstructed AST.

Chapter 4

Framework Design and Implementation

In the previous chapter we described what a system based on our generic framework will do. The purpose of this chapter is to analyze how the system performs the specified task using the new framework. The major design issues which had to be resolved, for this generic framework, are :

- the low level representation of the AST and the query-concept description,
- the Abstract Concept Language,
- the main code localization algorithm,
- meaningful result forms and
- human interaction with the system.

We must remind the user that our most important constraint was the second core requirement specified in the analysis phase: the system should be capable of performing its main task with inputs expressed in different languages with minimum-effort changes in the code using the same framework. In the following sections we describe the adopted design strategy to resolve all the main issues mentioned.

4.1 Code and Query low level representations

Every language consists of a set of basic constructs. In C for example we can have iterative or conditional statements, in HTML on the other hand a paragraph or a sentence can be considered a basic construct. Using a domain model for each language which captures the language's basic constructs and features, it is possible to abstractly represent "source code" in this language.

We call domain model a set of classes that capture these primitive-basic constructs of a language. Using the domain model adopted for the REVENGE project, "source code" is parsed in Ariadne and an annotated AST is constructed. Each node of the AST is an instance of a class defined in the domain model. Examples for the C language can be : a `Function_Definition` cl^ss or an `If_Statement` class.

Domain models are treated as hot spots in our framework. For each possible target language for the system a new domain model should be created. The most difficult part in creating the domain model is to identify the crucial basic constructs of a language and possible abstractions of them. Virtual functions that need to be implemented in the base classes of a new domain model will present similarities to the ones implemented for the C domain model. As a result we expect the necessary amount of effort required to come up with a domain model for a new target language to decrease significantly for any subsequent target language.

Assuming we have a parsing facility for the new target language, like the one provided by Ariadne for C, one can use its domain model to create an AST for "source code" in this language.

Our system receives a description of the nodes of the AST created by Ariadne in s-expression format and then reconstructs a simplified AST using a subset of the original domain model. If the target language has few basic constructs then adopting the whole domain model for concept recognition is not a problem. In cases like C or Pascal, which have large domain models, we can perform the task of concept recognition using a "lighter" version of the domain model which "ignores" some classes by keeping their superclasses. The analyst can still refer to the missing classes by using their superclass thus achieving abstraction which is a key concept in design recovery. When choosing which classes can be omitted one should remember that a certain degree of expressiveness is necessary in order to be able to have a meaningful intermediate representation. The designer should make a compromise between a "lighter" - easier to use domain model and a more expressive but less abstract domain model. As we saw in chapter three the "source code" and query-code description given by the analyst use the same low level representation.

Having the previous observations in mind, we will now describe our framework for the "source code" and query low level representations. The system implemented using this framework accepts C code and thus all the examples from here on will be based on C and for some of them we will show possible extensions with other languages.

The basic framework superclass is called the `State` class and serves as the superclass for the classes in all domain models. The `State` class captures the necessary common attributes of all classes in a domain model. It has for example an identification attribute in which an identification string for every node in our AST is stored and a type attribute used to indicate the domain model a descendant of this superclass belongs. The `State` class also defines several virtual functions, implemented differently in every language domain model (i.e. the *traverse_tree* function which traverses the reconstructed source code AST).

Each domain model should have one superclass which captures the common attributes of its descendants for the specific language, for C we call this superclass the `C_State` class. Such a superclass serves mainly as an abstract class capturing features and functionality common to all classes in its domain model. Member functions of this superclass are mostly related to the code localization process. Descendants of this superclass are all the classes in the domain model representing the basic constructs of the language or their abstractions (i.e. For_Statement class, Iterative_Statement class).

In order to describe and recognize a code segment or a concept we rely on some formal, structure oriented pattern of features. The next few paragraphs describe how our framework captures possible features. Every language might introduce its own features, we can recognize two categories of features : features common to all classes in the domain model of a language and features particular to some classes only in the domain model of the language. A class called Feature serves as an abstract class for all classes describing features in any language.

For the C language we define a new class called C_Feature which is derived from the Feature abstract base class and acts as a feature container class (see figure 4.1). Any object can have a number of features which are stored in a list. Each element of this list (i.e. a feature) belongs to a class called the Feature_Item class. Four classes describing features common to all C basic constructs, namely Uses_Description, Defines_Description, Keywords_Description and Metrics_Description. Instances of the Uses_Description class store the variable names used in a basic construct. Defines_Description objects store the variable names set in a basic construct and instances of the Keywords_Description class store all identifiers occurring anywhere in the basic construct. Finally objects of the Metrics_Description class capture the values for the five metrics calculated by Ariadne for a basic construct. A feature unique to only one class in the domain model will appear as an attribute of this class.

If we wish to add a new feature for a language we just have to create a new class for it, make this class a descendant of the Feature_Item class, and update the feature comparison algorithm to include the new feature. If we introduce a whole new language then in its domain we must specify a new abstract feature class (HTML_Feature for example) and then define classes for its new features which will be subclasses of the Feature_Item class. The feature comparison algorithm depends on the low level representation of the features (i.e. simple string comparison) and can be the same as the one used for C or altered depending on the form of the new features. Classes in resulting framework are grouped in libraries and can be reused and incorporated to new systems.

In figure 4.1 we show part of the framework used for the low level representation of the AST and the code segment abstract description using the Booch notation described in [6].

4.2 Abstract Concept Language

In order to retrieve code segments based on the function they perform, a concept language is introduced and used as a query language. This language can be either generic, so that it could be used for any programming language, or specialized for each target language. It is our belief that in order to be able to capture the most important features of various languages (e.g. HTML, C, Pascal) a specialized concept language for every target language should be created. Thus the creation of a concept language is a hot spot in our design. Languages like C and Pascal might of course share the same concept language, or parts of it, as they resemble semantically and syntactically . For reasons well known in Information Retrieval, partial matching should be possible when queries are formulated with such concept language.

Going through the literature one can see that there is no consensus on the way a language capable of describing concepts should be created. Our experience with the Refine prototype was reported in [26], the elements of an Abstract Concept Language (ACL) we deem necessary are:

- abstract statements (S) able to describe all basic language constructs,
- don't care statements (DCS) that can match any language construct and

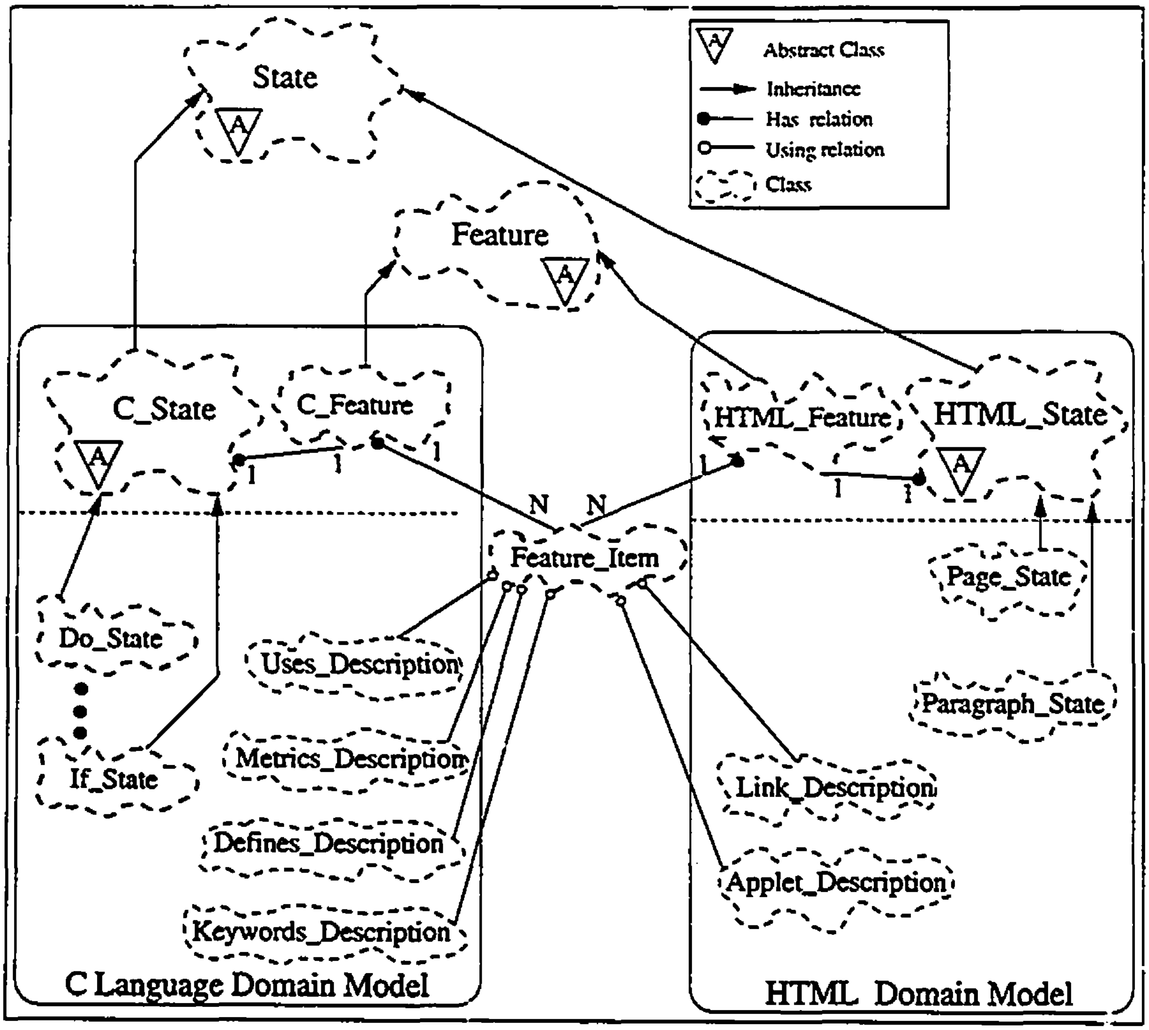


Figure 4.1: Main system class design.

- macros (M) to facilitate hierarchical plan recognition [13].

We consider these to be the minimum requirements for a sufficiently expressive concept description language. The language can also be adorned with typed variables, operators or any other features the developer judges useful.

Don't care statements are necessary because they can be used as "gluing" material among fine-grained abstract concept descriptions in order to express a larger-grained concept (*hierarchical recognition*). In our implementation for the C language we provide three don't care mechanisms in the form of two abstract statements:

1. the **-Statement*,
2. the *+-Statement* and
3. the *empty* feature value.

The *empty* feature value denotes a match with any feature vector obtained from a candidate code fragment to be matched. The **-Statement* will match zero or more code segments of any type, while the *+-Statement* will match one or more code segments of any type. If the analyst specifies features for these don't care statements then only code segments of any type which have these features will be recognized.

Existence of macros in the language allows the analyst to refer to plans to be included at parse time in a query, in order to describe a larger concept. For example the analyst can say :

SOURCE : another-plan-filename

inside a query. This will result in inclusion of the plan, described in the file with the specified filename, inside the currently described plan. An example of a query in ACL for C follows:

```
Iterative-Stmt
    abs-exp-desc
            keywords : [?element]
   {
    *-Stmt
     abs-gen-desc
              empty;

    Assignment-Stmt
     abs-gen-desc
              uses : [ list,?element],
              defines : [head,?element]
  }
```

Using this quety we can locate all iterative statements which have an assignment as the last statement in their body. We also specify that the assignment statement should use a variable called list, define a variable called head and both use and define a variable, which should also appear in the condition of the iterative statement and has the symbolic name, ?element.

The use of query variables (identifiers preceded by a question mark also called bind variables) is a feature we found quite useful in our prototype and which is also part of our implementation of the ACL for the C language. A more detailed presentation of ACL as well as several example queries, can be found in appendices A and B.

4.3 Main code localization algorithm

Based on the requirements and decisions analyzed in previous paragraphs the design of the main CSL module was completed. In Figure 4.2 we present a high level scheme of the Code Segment Localizer module.

The input to the CSL module, as shown in figure 4.2, is the location of two files. The first file is the collection of s-expressions describing the AST for the source code. The second file contains the abstract description of the code segments we want to locate expressed in ACL. The AST description file is passed to the s-expression parser submodule which parses the file and creates an object for each s-expression in the file.

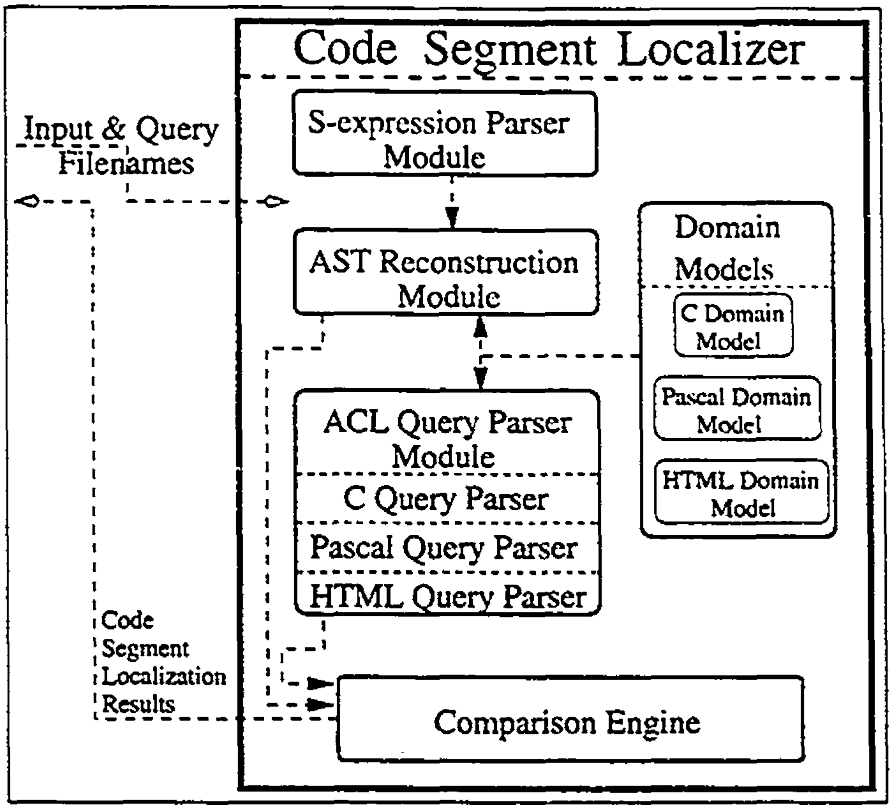


Figure 4.2: The CSL Module.

These objects are then passed to the AST reconstruction submodule. The purpose of this submodule is the creation of the source code intermediate representation for our tool which is again an AST. We will use the T_c symbol to refer to this AST. To create the T_c AST we need to use the classes in the domain model of the target language.

The AST reconstruction module works in the following way, the s-expression file describing the source code is parsed and for each s-expression a generic object is created. The resulting objects are stored in a "flat" linked list. In the second phase of the AST reconstruction process starting from the *Function Definition* objects we build the sub-AST for each function in the system and at the end we gather all these sub-ASTs in a linked list which is the simplified AST we are going to use for our computations.

The creation of sub-ASTs is rather interesting, starting from a generic object describing a *Function Definition* object we create a new object using the constructor of the corresponding class from the language domain model (i.e. `Function_Def_State` class). The next step after the creation of any object in the AST is to scan its "source" generic object for attribute values and update the attributes of the new object. If some attribute value is a reference to another generic object then a binary search algorithm is used to locate the referenced generic object in the linked list and the process of object creation and update is invoked recursively. For example after updating the simple feature values of a `Function_Def_State` class we have to set the *function body* attribute of the object; this attribute is a reference to another generic object with a unique id. Using this unique id we retrieve this generic object and create a new "specific" object depending on the generic object's type. The generic object's type is specified as the value of its base class in the domain model used to create the original AST. Having adopted a lighter version of this domain model we can map all the originally used classes to some class in our domain model.

For large systems the AST reconstruction process is by far the most expensive time wise. Let N_o be the total number of s-expressions describing objects and N_a be the total number of attributes of these N_o objects, then $N = N_a + N_o$ is a good approximation of the total number of objects in our final AST. The cost of the creation of each intermediate object is $O(1)$ (just a simple sequential read from a file). The creation of the final AST object from its intermediate representation will cost at most the number of the object's attributes multiplied by $\log N_o$, because $\log N_o$ is the cost of a binary search in the sorted list of intermediate objects already created. Thus the worst case cost would be a binary search for every intermediate object for all of its attributes, this bounds our algorithm to be $O(N_o + N_a \log N_o)$. In reality the algorithm is much faster as it takes out of the remaining object list any object that is located through the binary search and corresponds to an attribute. We are currently considering the possibility of avoiding the creation of intermediate (generic) objects in order to speed up the whole process.

The file describing the query is parsed by the ACL query parser submodule. The result of the parsing is again the creation of an AST (T_a). Creating the T_a AST requires the use of the classes in the same domain model used for to form the T_c AST.

Both ASTs are then passed to the comparison engine submodule that performs the actual localization task. Figure 4.3 shows a simplified view of the T_a and T_c ASTs formed for the query presented in the previous section and a possible "matching" piece of code.

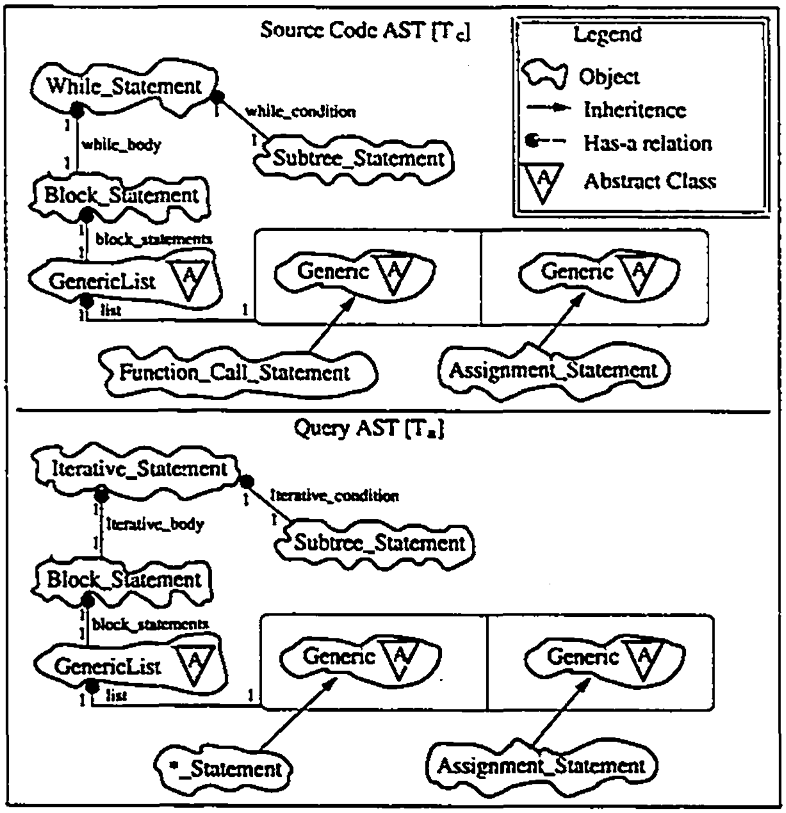


Figure 4.3: Example T_a and T_c ASTs.

The algorithm used to match an abstract pattern described in ACL with the intermediate representation of our "source code" is essentially the one described in [26]. We will analyze the algorithm and focus on how it was mapped on our object oriented framework.

The main steps of the algorithm after the creation of the ASTs are :

1. creation of a StatiC Model (SCM) specific to the target language domain model,
2. creation of a Markov Model from the ACL AST (T_a),
3. selection of candidate parts of the code to serve as initial points for the localization process and finally
4. invocation of a Viterbi [68] algorithm to find the best fit between the code segment described and an actual code sequence starting at a candidate point.

4.3.1 The StatiC Model (SCM)

The SCM is a simple automaton that shows the possible decomposition of abstract classes and "quantifies" the analyst's belief about the ability of the abstract class to "generate" a particular source code segment. A part of the SCM for the C language is shown in figure 4.4. As we can see an object of the `Iterative Statement` class can be decomposed, or simply allowed to match, any of the three classes (i.e. `For_Statement`,`Do_Statement` and `While_Statement` classes), specified by the SCM. Every possible decomposition is assigned a probability

$$P_{SCM} = P_{SCM}(S_i|A_j)$$

where S_i is a source code statement (i.e. For Statement) and A_j is an abstract statement description in the ACL query (i.e. Iterative Statement), indicating the analyst's belief about its possibility of appearing in the code. This probability can be :

- given by the programmer as part of the query,

- supplied by the system using a uniform distribution based on the number of choices (current implementation) or
- it can be calculated dynamically at run time based on the matches obtained so far.

These probabilities on the SCM are used later in the calculation of the concept-to-code distance or similarity measure and can be easily changed if necessary. In the initial implementation of the algorithm, the SCM was also used for type checking. The new implementation does not rely on the SCM for general type checking.

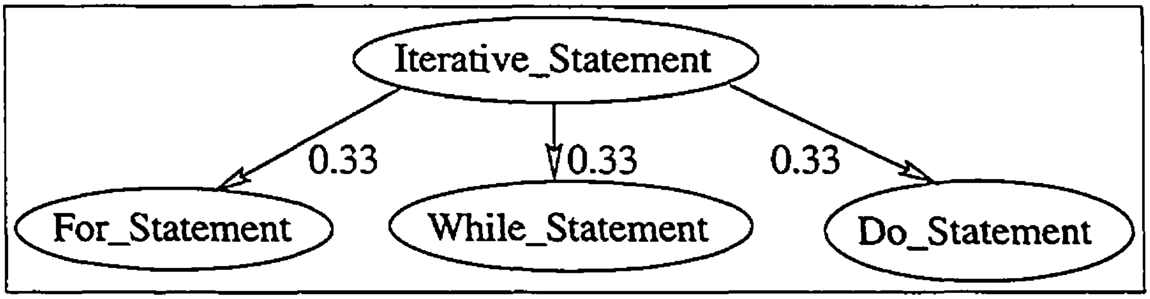


Figure 4.4: Part of the SCM describing the Iterative Statement "decomposition".

4.3.2 The pattern matching process

The following three sections describe in detail the core methodology used to perform pattern matching of features among nodes in the T_c and T_a ASTs.

Markov Model creation

The existence of abstract (e.g. the *Iterative Statement*) and don't care statements (e.g. **-Statement*, *+-Statement*), in our ACL, allows generation of many possible code segments from a given query expressed in ACL. Markov models provide an appropriate mechanism to represent these alternatives [52].

A Markov Model is a source of symbols characterized by states and transitions. Two special states exist: the starting state and the final state. The starting state has no incoming transitions and the final state has no outcoming transitions. A model can be in a specific state with a certain probability. Each state has a finite number of transitions leading to other states each associated with a certain probability. Transition from one state to another state can only happen when a "symbol" associated with a valid transition is recognized and "consumed". Generating a Markov Model for the query AST T_a allows the subsequent use of the Viterbi algorithm to calculate the sequence of transitions which maximizes the total probability of a path beginning at the starting node and ending at the final node of the model. The path corresponds to the matching between T_a and T_c.

Using the query's AST (T_a), the Markov Model is created dynamically by simply traversing the AST, the building algorithm is simple. A transition is allowed and added from each basic construct description node to the next node in the AST. Star and plus statements (**-Statement*, *+-Statement*) need special handling. Each of the latter statements always has an outcoming transition which returns to itself. Also statements preceding a **-Statement* should have additional transitions to the statement following the **-Statement* (see figure 4.5).

We call the resulting Markov Model: Abstract Pattern Model or simply (APM). The APM is actually implemented on top of the query's AST by adding possible transitions to the nodes of the T_a AST. That is the reason we refer to classes in the domain model as *States*, as they also represent actual states in the APM.

An example of a simple APM is shown in figure 4.5, elements of the T_a AST are omitted on purpose in order not to clutter the figure. For composite statements (i.e. an *If Statement* with *then* and *else* parts) the process of creating the APM is invoked recursively. Each transition has an associated probability; all transition probabilities are initialized to -1 before the matching process and this is the reason we chose to omit then in figure 4.5.

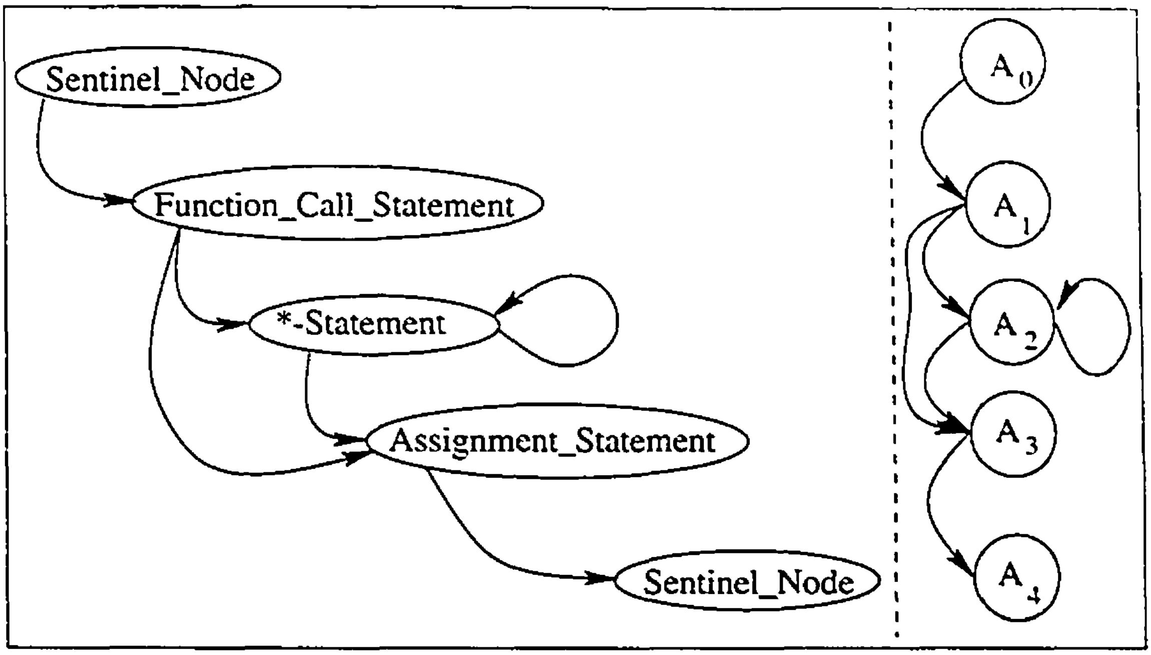


Figure 4.5: Example of dynamically created APM.

The localization algorithm

The first step for the algorithm is to locate the candidate starting points in the source code AST T_c, this task is also known as *source code segmentation* or *code delineation* and the algorithm used is the one described in [26]. The *code delineation* algorithm has two main steps, first we locate all possible starting points based on generic criteria (i.e. type compatibility) and then we refine the initially selected set of candidates by performing a series of feature comparisons. In our implementation this second step of the source code segmentation process is the initial step of the main localization algorithm.

For the first step of the segmentation process we choose the first "concrete" statement S ("concrete" means that S can not be a don't care statement) from our query and locate all occurrences of statements which are type compatible with S everywhere in T_c. In order to ensure that all possible candidate points will be considered a generic check is used in this phase (i.e. type compatibility between the first "concrete" statement in the query and a node in the source code AST) while traversing recursively the source code AST. There are however some special cases. If the query consists only of various don't care statement then we return all the logical blocks as possible starting points, this decision was taken after careful consideration of the most meaningful queries that can be constructed solely from don't care statements.

Dynamic Programming match between concept and code ASTs

At this point we have all the necessary input for our main localization algorithm. The Viterbi dynamic programming algorithm is used to find the path that maximizes the overall generation probability among all the possible alternatives formed by the APM created for a given query. In the next paragraphs we describe the algorithm.

Let $S_1, .., S_k$ be a sequence of program statements (represented as objects of the T_c AST, occurring at a certain candidate starting point in our "source code") and $A_1, .., A_n$ be a possible sequence of states (also represented as objects of the T_a AST) in our APM. Then a possible recognition sequence would be of the type:

$$\underbrace{S_1, .., S_{g_1}}_{A_1}, \underbrace{S_{g_1+1}, .., S_{g_2}}_{A_2}, ..., \underbrace{S_{g_{i-1}+1}, .., S_{g_i}}_{A_j}, ..., \underbrace{S_{k-1}, .., S_k}_{A_n}$$

meaning that abstract statement description A_1 matches statements : S_1 .. S_{g_1}, abstract statement description A_2 matches statements S_{g_1+1} .. S_{g_2} and so on. We call statements S_{g_1}, S_{g_2}, ... ,S_k breakpoints.

The purpose of our algorithm is to find the most likely statement sequence $S_{g_{i-1}+1}$, .., S_{g_i} that contributes to maximum similarity when combined with similar matches of other states.

The matching process for a single statement and its abstract description can be broken down into three discrete checks, failure in any of these steps terminates the comparison process for the current starting point and causes a transfer to the next possible starting point. Failure usually means that the probability computed is less than a user-specified threshold. The three checks performed are :

1. type compatibility check,
2. metric proximity check and
3. feature vector value comparison.

These steps performed for a candidate starting point are actually the second step of the *code delineation* process described in [26]. The metric proximity check can be used when the comparison granularity is at the level of a `begin-end` block; the formula used is described in the following section. For statement level granularity we use dynamic programming techniques to calculate the best alignment between two code fragments based on *insertion, deletion* and *comparison* operations. Rather than working directly with textual representations, source code statements are abstracted into feature sets that classify the given statement. The whole process is described in detail in a following section (i.e. section 4.3.4). Dynamic programming is a more accurate method than the direct metric comparison based analysis because the comparison of the feature vector is performed at the statement level.

Checking type compatibility is accomplished using information in the domain model and the SCM if necessary. The possible result is a boolean value indicating if the statements checked have compatible types. Statement type compatibility is given

- for simple statements: by comparing the type attribute of each object in the query and the source code AST or
- by using the SCM if the query object is an instance of an abstract statement class (i.e. `Iterative_Statement` class).

The euclidean distance of metrics is calculated and used as a comparison factor in cases where the metrics are specified for the abstract description. The distance calculated should be less than a certain threshold which can be set by the analyst. The euclidean distance C is calculated using the following formula :

$$C(\mathcal{P}_i, \mathcal{S}_j) = \sqrt{\sum_{k=1}^{5} (M_k(\mathcal{P}_i) - M_k(\mathcal{S}_j))^2} \qquad (1)$$

Where $\mathcal{P}_i$ is the i-th statement after the current starting point in the "source code", $\mathcal{S}_j$ is the j-th statement described in the query and $M_k(S)$ is the k-th metric value for a statement S. To compute C we use the values of the five metrics computed by Ariadne. If no metrics are specified in the abstract description of a statement in the query then this check is omitted.

The result of feature comparison is a similarity measure of the segments being compared. If S_i is a composite statement then recursive calls of the functions performing feature comparison take place.

If for example S_i is a while statement, first a type compatibility check with its possible description A_j occurs. The next step is to calculate the euclidean distance C between the metric values of A_j and S_i, using the previous formula, and then compare C with the given acceptable threshold for metric distance. Absence of metrics for A_j is interpreted as a don't care value. Finally the similarity measure produced by the feature comparison for the while statement itself is "combined" with the similarity measures produced by recursive calls to the matching functions for:

- the expression used in the while condition and
- the statement describing the body of the while loop

to produce the overall matching probability. The calculation of the similarity measure is described in the following paragraphs.

Similarity measure

Assuming a match between a sequence of source code statements $S_1, .., S_k$ and a sequence of abstract code descriptions $A_1, .., A_n$ we need to compute a measure of our belief for this potential match. For convenience let us use the same recognition sequence as before :

$$\underbrace{S_1, \dots S_{g_1}}_{A_1}, \underbrace{S_{g_1+1}, \dots S_{g_2}}_{A_2}, \dots, \underbrace{S_{g_{i-1}+1}, \dots S_{g_i}}_{A_j}, \dots, \underbrace{S_{k-1}, \dots S_k}_{A_n}$$

What we actually try to match is objects in the two ASTs (T_c and T_a). Thus a possible measure of similarity between T_c and T_a can be the following probability:

$$P_r(T_c|T_a) = P_r(r_{c_1}, \dots r_{c_i}, \dots r_{c_I}|r_{a_1}, \dots r_{a_j}, \dots r_{a_J}) \qquad (2)$$

where, $(r_{c_1}, \dots r_{c_i}, \dots r_{c_I})$ is the sequence of grammar rules used for generating T_c and $(r_{a_1}, \dots r_{a_j}, \dots r_{a_J})$ is the sequence of rules used for generating T_a. We will use an approximation of this formula.

Using the Viterbi dynamic programming algorithm and the created APM we can compute the probability:

$$P_r(S_{g_{i-1}+1}, .., S_{g_i}|A_{f(i)}) \qquad (3)$$

where

$$S_{g_{i-1}+1}, .., S_{g_i}$$

is a sequence of statements in T_c that can be matched by the valid at the i-th comparison step abstract description $A_{f(i)}$. To find possible alternatives for $A_f(i)$ one has to calculate the reachable transitions in the APM at the i-th comparison step, this is represented by the subscript $f(i)$. In order to be able to match several actual code statements $A_f(i)$ must be a : don't care statement (i.e **-Statement* or *+-Statement*), a composite statement or a macro.

Using (3), an approximation of (2) is possible [29, 26]:

$$P_r(T_c|T_a) \simeq P_r(S_1; ..S_k|A_1; ..A_n) \simeq$$

$$max_{g_1,..,g_k} \prod_{i=1}^{k} P_r(S_{g_{i-1}+1}, .., S_{g_i}|A_{f(i)}) \qquad (4)$$

Formula (4) is essentially the result computed by the Viterbi algorithm. If $A_{f(i)}$ is a reachable state in the APM at the i-th step, then:

$$P_r(S_{g_{i-1}+1}, .., S_{g_i}|A_{f(i)}) = \prod_{l=g_{i-1}+1}^{g_i} P_r(S_l|A_{f(i)}) \qquad (5)$$

In the case of a composite statement, a Markov model is considered for it and is used in a similar way with the Viterbi algorithm. In general, the probability $P_r(S_i|A_j)$ has to be computed.

The resulting probability expresses the belief that the code segment S_i in our source code AST (T_c) can be described by the abstract statement $A_{f(i)}$ in the query AST (T_a). The actual value of the probability P_r for two statements is calculated by multiplying the probability for the abstract description statement defined in the SCM and the value we get from the feature comparison of the two segments. The feature comparison formula is presented in the next paragraph.

Feature Comparison

The features the analyst chooses to examine depend mainly from the analysis he is interested in. For the purpose of the analysis we perform in Ariadne we selected four features. The set of adopted features, for a C language statement S in our system, consists of:

- the set of variable identifiers defined in S ($\mathcal{D}$),
- the set of variable identifiers used in S ($\mathcal{U}$),
- the set of identifiers-keywords appearing in S ($\mathcal{K}$) and
- a set of five real numbers which are the values for the five metrics calculated by Ariadne.

Metrics comparison is used, if metrics are specified in the abstract description, as an initial testing step. If the euclidean distance calculated is bigger than a user specified threshold the segments are considered different and the comparison process stops.

Let A_j be a simple (i.e. non-composite) abstract description of statement S_i in the T_a AST, then the probability $P_r(S_i|A_j)$ in (5) can be calculated as follows:

$$P_{comp}(S_i|A_j) = \frac{1}{v} \cdot \sum_{n=1}^{v} \frac{card(AbstractFeature_{j,n} \cap CodeFeature_{i,n})}{card(AbstractFeature_{j,n} \cup CodeFeature_{i,n})} \qquad (6)$$

We chose three features for C ($\mathcal{D}, \mathcal{U}$ and $\mathcal{K}$) and thus $v = 3$ in the above formula. The total probability is equal to the sum of three fractions. Each fraction for $\mathcal{D}$, $\mathcal{U}$ and $\mathcal{K}$ is computed as the number of common identifiers for each pair of code segment-query segment, divided by the number of the total different identifiers for this pair. The final similarity measure for each transition

$$P_r(S_i|A_j)$$

can then be computed as:

$$P_r(S_i|A_j) = P_{comp}(S_i|A_j) \cdot P_{SCM}(S_i|A_j)$$

A full blown recognition example is presented in appendix C, the whole process we just described is explained in detail using a typical query.

4.4 Result form

In the case of successful recognition of a piece of code abstractly described in the query the analyst gets as an answer a set of locations in the source code for each abstract statement description in the query in the form :

filename: starting-line, ending-line.

For each occurrence of the concept reported the system also outputs the overall similarity measure calculated. The analyst can then manually inspect the code to determine false alarms.

If in our ACL query we described an assignment statement followed by a for statement then a matching piece of code would be reported as follows :

```
MATCH PROBABILITY : 0.34
                 MATCHING CODE
LOCATED IN : sa.c:1787,1788 is      ExpressionStatement_7364
LOCATED IN : sa.c:1787,1788 is      ForStatement_7365
```

The strings following the location of the code (i.e. ExpressionStatement_7364, ForStatement_7365) are the unique ids that identify the s-expressions used to describe the matching source code.

4.5 Human interaction with the system

The analyst interacts with the system through an intuitive and extensible graphical interface. Using the interface the analyst can perform three operations:

1. create a query using a graphical or a textual editor,
2. adjust threshold values used by the localization algorithm and
3. inspect the reported results.

The GUI module is implemented in Tcl/Tk and can easily be extended to achieve greater functionality. An on-line help facility, in the form of explanatory balloons, helps novice users to explore the interface.

4.6 System architecture

A generic view of the architecture of the main system modules is presented in the next figure.

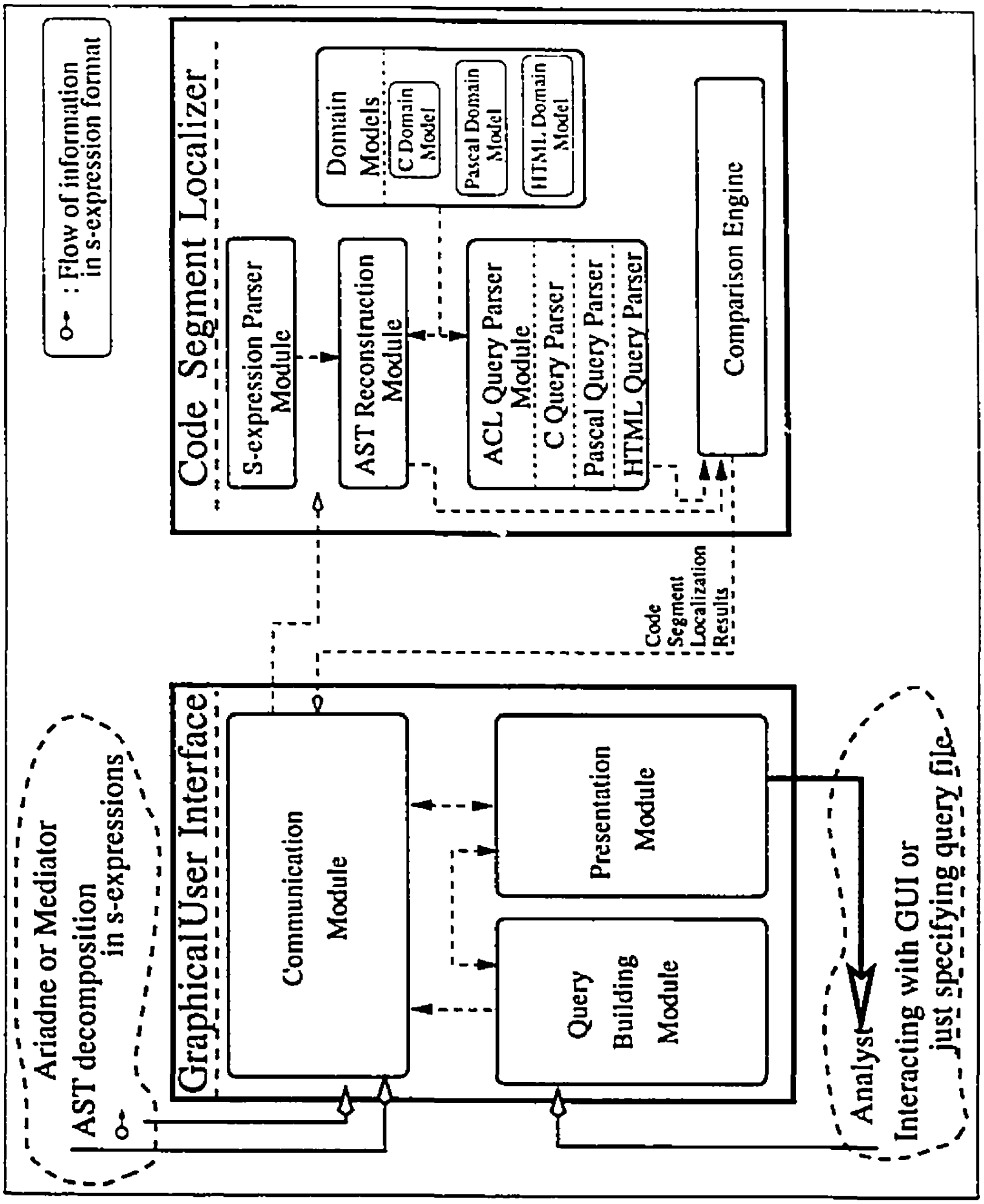


Figure 4.6: Generic architectural view of the system.

The architecture of the CSL module has been already analyzed. In this section we will briefly analyze the architecture of the GUI module and focus on the details of the comparison engine submodule in the CSL.

4.6.1 The graphical user interface

The GUI module can be decomposed into three submodules :

1. the communication submodule,
2. the query building submodule and
3. the presentation submodule.

The communication module is implemented using the Expect package under Tcl/Tk. When the GUI starts, this module takes control of the input and output channels of the C++ program implementing the CSL module. All message and data exchange between the GUI and the CSL module is performed using functions in the communication submodule.

The query building submodule consists of a graphical and a textual editor. The analyst can use either or both of these editors to create a new query. This submodule is implemented using the Tix package under Tcl/Tk. The graphical editor allows the analyst to write queries without prior wide knowledge of the grammar of ACL.

Finally the presentation submodule is built in and contains the necessary functions that implement all graphics used in the user interface.

4.6.2 The comparison engine

Our initial idea and objective for the design of the comparison engine was to introduce abstract base classes and virtual operations so that the comparison algorithm would be dynamically determined at run time based on the type of the entities compares (i.e. Pascal programs, HTML pages, C programs). However the need for a comparison function specific to some statements still exists because certain statement (classes in the domain model) can have unique features. For example to compare two objects that belong to the Function_Definition class we need to compare not only their standard features but their function names as well, the function name in this case is the specific feature that has to be compared. Standard feature comparison is implemented using one set of function for all the classes in a particular domain model. Specific feature comparison is done using specialized member functions in the class that defines this specific feature (i.e. Function_Definition class).

The design decisions adopted concerning the "distribution" of the algorithm among the classes are:

- gather all generic functions (e.g. *start_pattern_match*, *perform_pattern_matching*) in a submodule (we call this submodule: the **Comparison Engine**),
- implement the language specific functions (e.g. *compute_probability*, *check_type_compatibility*, *traverse_tree*)as member functions of the generic state class in the language's domain (e.g. *C_State*, *HTML_State*),
- implement comparison operators for all classes having special features.

Following the first decision a new class was created and named : Pattern Match Engine class. The main goal was to implement member functions for this class capable of performing all the generic steps of the algorithm. If we could achieve this the class could be used for all target languages for which we have specified a domain model. Thus member functions of this class would implement the core of our algorithm. The Pattern Match Engine class uses the StatiC Model (*SCM*) to retrieve the P_SCM probability (see section 4.3.1) and the two ASTs (i.e. query and source code ASTs). They key idea to keep the Pattern Match Engine class as generic as possible is to achieve AST manipulation through a common standardized interface, this is achieved by allowing communication only with the State abstract class which defines this uniform interface for all classes in any domain model (see figures 4.1,4.7).

Functions critical to the localization process are called from member functions in the Pattern Match Engine class. These critical function handle - among other things - the feature comparison, the selection of candidate starting points and the calculation of the overall similarity measure. All these functions are hot spots in our framework design, and as a result they will have a different implementation for each target domain - language. Consequently if the source code is in Pascal the functions defined in the domain model created for Pascal will be invoked where as if the analyst focuses in C programs the appropriate function in C's domain model will be used. The binding is done dynamically in every case.

Moreover the maintainer can define more than one functions to handle the above tasks in each domain and choose which one to use at run time (plug and play capability). Currently, for example, we have two ways of doing the feature comparison, the one described previously and a simpler method that we use for testing and validation. The analyst can define in the command line or at run time which one he wants to use every time.

Figure 4.7 shows how the Pattern Match Engine class communicates with the two ASTs (T_c and T_a) through calls to virtual member functions of the State class. Classes in any new domain model have to respect the interface defined in their abstract superclass (i.e. the State class). Using dynamic binding functions in the Pattern Match Engine class will invoke the correct function for the corresponding domain model every time.

The two most critical member functions of the*Pattern Match Engine* class are:

1. the *start_pattern_match* function and
2. the *perform_pattern_match* function.

The *start_pattern_match* function is responsible for the initial steps of the algorithm, it calls a function to find all candidate starting points and then performs a

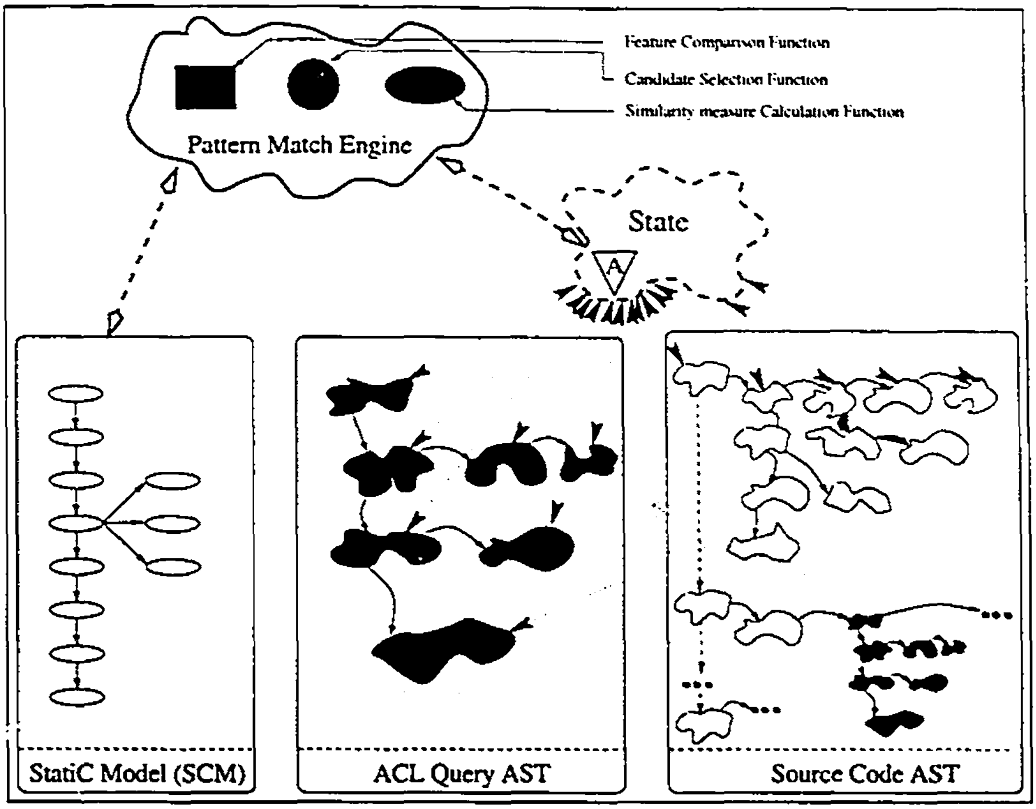

Feature Comparison Function
Candidate Selection Function
Similarity measure Calculation Function
Pattern Match Engine
State
A
StatiC Model (SCM)
ACL Query AST
Source Code AST

Figure 4.7: Simplified interaction diagram for the Pattern Match Engine class.

loop over all the possible starting points calling the *perform_pattern_match* function for each one of them.

In the *perform_pattern_match* function we traverse the T_a AST, using the APM, and the T_c AST, and then call a language specific function (hot spot) to compute the similarity measure of the active nodes in the two ASTs for every step of our traversal. If we reach a final node in the APM then our comparison was successful and the location and the total similarity measure are returned; if not, failure is reported.

The State superclass (see figure 4.1), used as an abstract class for all domain models, defines virtual functions implementing several parts of the localization algorithm . The actual implementation of these function is located in the language specific superclass (e.g. *C_State*, *HTML_State*). The *C_State* abstract class implements functions to:

1. check for type compatibility,
2. compute the euclidean distance of the five metrics using formula (1),
3. manipulate the domain model specific SCM,
4. calculate the similarity measure using formula (4) and
5. calculate the similarity measure for composite statements.

All these functions are implemented for the C language specific domain model. If we choose another target language then in its domain model we should define similar comparison functions which are specializations of the virtual functions (or operators) defined in the `State` class.

Finally for each class in the domain model a function called *match_specific_features*, declared as virtual in the domain model superclass, is implemented to match unique features of a class with their description in the APM state. For example an instance of the *Function_Call* state class will define the name of the function called by the code segment it describes; this is considered a unique feature and its comparison is handled by the implementation of the *match_specific_features* function for the *Function_Call* state class.

A view of the design described in this section in the form of a class interaction diagram is shown in figure 4.8. In the diagram the reader can see the message exchange between classes in the system. Note that messages in C++ are actually function calls to class member functions.

4.7 Evolution and Maintenance

The design reported in the previous section is the result of several iterations over the initial requirements and ideas for plausible designs and their implementations. Chronologically, the s-expression parsing module was build first, followed by the the

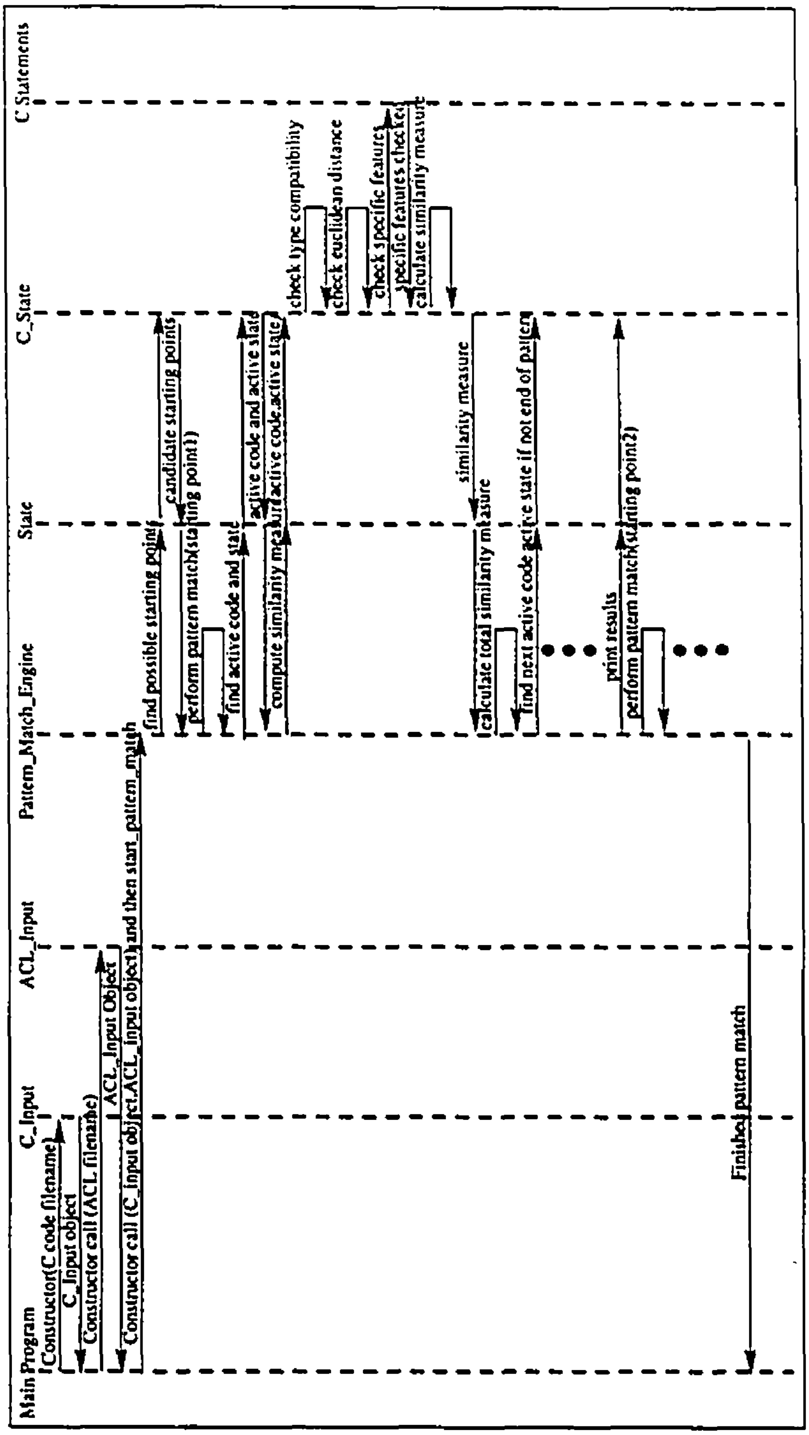

Main Program
C_Input
ACL_Input
Pattern_Match_Engine
State
C_State
C Statements
Constructor(C code filename)
C_Input object
Constructor call (ACL filename)
ACL_Input Object
Constructor call (C_Input object,ACL_Input object) and then start_pattern_match
find possible starting points
candidate starting points
perform pattern match(starting point1)
find active code and state
active code and active state
compute similarity measure(active code,active state)
check type compatibility
check euclidean distance
check specific features
specific features checked
calculate similarity measure
similarity measure
calculate total similarity measure
find next active code active state if not end of pattern
print results
perform pattern match(starting point2)
Finished pattern match

Figure 4.8: Simplified system interaction diagram.

	Number of Functions	*LOC*	*Number of related classes*
S-expression parser	5	5 K	2
AST reconstruction module	155	3K	26
Domain Model (C)	240	8 K	32
Comparison Engine	24	3 K	3
ACL Query parser	6	3 K	2
Total	430	22 K	65

Table 4.1: Module sizes.

domain model for C and the AST reconstruction module. The implementation of the ACL parser was the next step. Finally the comparison engine was built and several member functions were added to the domain model classes in order to complete the localization algorithm. The GUI module was created after a reasonably stable version of the system was available. Table 4.1 presents some approximate numbers related to the system's size.

Recent work explores mainly two topics:

1. possibilities to improve the algorithm by introducing new low level comparison methods and
2. adapting the design to accommodate new domains (i.e. HTML, structured text).

Ideas and work to achieve these goals are reported later in chapter six section one. Our experience during evolution and maintenance indicates that our approach for the system's design was robust. Additional functions are easy to incorporate and most importantly debugging is fairly easy because of the modularity achieved.

Chapter 5

Experimental Results

This chapter discusses results obtained from our experiments. The first section briefly describes the subject systems we used for testing the capabilities of the tool. The next section focuses on the description of some concepts or plans we used. Finally in the last section we present and discuss the results of our experiments.

5.1 The Subject Systems

Testing a design recovery tool presents a major difficulty, the developer has to play the role of the analyst and recognize concepts in a subject system; to be able to validate the output of the tool the analyst must have a good knowledge of the subject system functionality and design. To overcome the above mentioned problem we adopted the following strategy. We chose as test cases :

- small size C programs for which we had complete knowledge of their design and functionality ourselves,
- medium size programs for which we had access to their developers and
- large modular programs.

The two small systems are : a simple linked list manipulation program (around two hundred Lines Of Code (LOC)) and a program simulating the popular card game "blackjack" (nine hundred and fifty LOC). These two programs although they are small and simple they contain a number of programming plans (i.e. list traversals, reading from files) as well as a number of "business rules" (i.e. how cards are dealt, what is the value of the cards).

For the medium size programs we chose two systems created by the speech recognition group in our lab. The first system is a speech decoder using the Viterbi algorithm on Hidden Markov Models (HMMs) [52]. The size of the speech recognizer (called simply Recognizer from here on) is around seven KLOC. The second system is the front end of a second speech recognition system developed in the lab. The system uses digitized speech samples as input to extract features relevant to the speech recognition task, we call this system the Feature Extractor. The Feature Extractor is around eight KLOC long. These two systems were selected because they contain a number of mathematical computations and were typical representations of a specific domain (i.e. speech recognition).

The choice of larger systems to be used as test cases was more difficult, we had to find systems modular enough to ensure that a certain concept can be found only in a small number of modules in order to facilitate validation of our results. Assuming this fact we did not have to have a perfect understanding of the whole structure and design of the system. To locate a concept we focused on one module, if the same concept was reported found in other modules during our tests we checked the validity of the result comparing the reported concept instantiation to the original concept used to create our query.

The first system chosen is NASA's C Language Integrated Production System or simply (CLIPS). CLIPS can be used as an expert system construction tool. We found CLIPS modular enough for our needs and also familiar because of our experience using it and analyzing it as a test case for Ariadne. Using Ariadne's analysis capabilities we had a fairly good knowledge of the system structure. The size of

Subject system	*Code size*	*Size of intermediate representation*
List	181 LOC	50 KB
Twentyone	942 LOC	322 KB
Recognizer	7 KLOC	2163 KB
Feature Extractor	8 KLOC	1014 KB
Clips	33 KLOC	8770 KB
Tcsh	45 KLOC	9661 KB

Table 5.1: Physical size of subject system and their intermediate representations

CLIPS is approximately thirty three KLOC. Finally we chose the popular Unix shell Tcsh (Cornell version 6.06) as our second large subject system. Tcsh was also used to test Ariadne and as a result we had a fairly good idea of its structure. The code for Tcsh is forty five KLOC long. Both these systems are modular enough for testing purposes and contain a wealth of programming patterns both generic and domain specific.

Although the size of our test cases might seem small compared to a multimillion line legacy system we believe that the design of our system can accommodate very large systems as well. The input to our system is not the subject system's code but an intermediate representation of it using the s-expression formalism. The input can be requested and sent from the global repository or generated and sent directly from Ariadne. The size of the files with this intermediate representation for the above systems is reported in table 5.1.

For very large systems the analyst can process the intermediate representation of the system module by module. Splitting the intermediate representation file is possible using a simple text editor or directly by requesting from the repository only s-expressions describing a specific system module. We have not encountered problems even with our largest subject system but we believe that for performance reasons it might be better if the user splits the system into several modules and then tests each one separately. The vital issue of scalability can be resolved using this technique.

5.2 Measuring performance

Using for all the subject systems the queries that reported the minimum and the maximum number of concept instantiations we obtained data regarding the time performance of our system. Results are presented in tables 5.2, 5.3 and 5.4.

By far the most expensive part, in terms of time always, is the parsing of the s-expression file, which describes the source code, and the reconstruction process that immediately follows the parsing. Impressive numbers were reported for the rest of the activities. All of the remaining reported activities involve mainly navigation through pointers which explains the reported - satisfactory results. Moreover, the most important part of our algorithm, the main localization and feature comparison process, performs very well even for our largest subject system (see table 5.4). Based on this latter fact we believe that the main localization algorithm can be successfully used for considerably larger subject systems.

5.3 Concepts and plans

This section analyzes our method of capturing and describing concepts used for our experiments. As the degree of our familiarity with each subject system varies we had to adopt different tactics for capturing plans.

For the smaller systems (i.e. List and Twentyone) full understanding of the code was possible. Going through the code we discovered several pieces of code which implement key concepts (e.g. the traversal of a list). Seeing the actual code segment which implements a concept the analyst can then use corresponding ACL abstract

	List Q1	*List Q2*	*Twentyone Q1*	*Twentyone Q2*
AST reconstruction	0.1 sec	0.1 sec	4 sec	4 sec
ACL Query parsing	0.1 sec	0.1 sec	0.1 sec	0.1 sec
Find candidates	0.1 sec	0.1 sec	1 sec	0.1 sec
Localize code	0.1 sec	0.1 sec	1 sec	0.1 sec
Candidates found	1	12	1	18
Concept instantiations	1	12	1	18
Stmts in Query	5	3	6	2

Table 5.2: Time statistics (part I).

	Recognizer Q1	*Recognizer Q2*	*F.Extractor Q1*	*F.Extractor Q2*
AST reconstruction	31 sec	30 sec	14 sec	14 sec
ACL Query parsing	0.1 sec	0.1 sec	0.1 sec	0.1 sec
Find candidates	0.1 sec	0.1 sec	0.1 sec	0.1 sec
Localize code	2 sec	3 sec	1 sec	8 sec
Candidates found	204	204	103	689
Concept instantiations	1	77	2	216
Stmts in Query	9	5	5	3

Table 5.3: Time statistics (part II).

	CLIPS Q1	*CLIPS Q2*	*Tcsh Q1*	*Tcsh Q2*
AST reconstruction	1366 sec	1367 sec	1547 sec	1552 sec
ACL Query parsing	2 sec	1 sec	2 sec	2 sec
Find candidates	100 sec	99 sec	312 sec	188 sec
Localize code	153 sec	248 sec	111 sec	196 sec
Candidates found	1890	1890	4209	2730
Concept instantiations	9	233	2	199
Stmts in Query	7	8	9	3

Table 5.4: Time statistics (part III).

statements to describe it. We found that in most cases the use of the graphical query builder speeds up the whole process significantly.

To locate and describe concepts for the medium size programs we relied mostly on their developers. We asked the developers to show and explain to us code segments implementing various key concepts. Moreover, we asked them to abstractly describe these concepts in terms of the query language and point all their occurrences in the code they were aware of. The final step was to refine the developer's concept description to make full use of ACL features.

Probably the hardest part was to identify concepts for the larger subject systems. To accomplish this task we relied heavily on system decomposition performed by Ariadne [21] and also on comments in the code itself. As we will see in the next section we need to know exactly how many times a certain concept occurs in the whole system to report meaningful precision and recall results. To overcome this obstacle we relied on the subject system's modularity and checked reported concept instantiations outside our target module manually.

5.3.1 Hierarchical concept formation and recognition

Hierarchical concept recognition refers to the ability of recognizing complex concepts from simpler ones. In this section we present an example of macro usage to move from fine-grain concept, or simple code segment, description to large-grain concept description and localization. To illustrate this we will use a concept from the Recognizer.

The Recognizer uses a Viterbi based algorithm on Hidden Markov Models to calculate a maximum likelihood transition sequence among the Markov Model states that represent phonemes. The following piece of code performs the calculation of a state's contribution in the resulting path. The first line initializes the total contribution of a transition to a constant minimum value. The loop starting in line 2 computes the contribution for all possible transitions between two states in the Markov Model. Initially (line 3) a check occurs to see if a probability for a certain transition has already been computed. If the result of the check is negative then the probability for this transition is computed (line 4) and a flag is set (line 5). The statement in line 7 checks if the calculated contribution is greater than the accumulated total contribution, if so it updates the total contribution value (line 8). The next line (9) advances the pointer to point to the next transition between the two states examined. Finally the last two statements (lines 11 and 12) keep track if a set of transitions has already been processed by setting an appropriate flag and using the accumulated total transition probability.

```
1       p = LOGZERO;
2       do {
3           if (!(distTested[idx = TrP->DistrIdx])) {
4              DistrVal[idx] = EvalDistr(&DistrList[idx],obs);
5              distTested[idx] = TRUE;
6           }
7           if(p<(contribution=TrP->Prob+DistrVal[idx]))
8              p = contribution;
9           TrP++;
10      } while(++i < *nextMix);
11      *mixTested = TRUE;
12      mix->Value = p;
```

Source code implementing a plan in a subject system [Recognizer]

The above presented code can be broken down to several smaller (fine-grain) concepts or plans. Based on their functionality lines 3 to 6 can be considered as one smaller concept (concept 1). The code segment starting from line 7 and ending at line 9 can be considered as a second plan and is called concept 2. Lastly the assignments in lines 11 and 12 implement another concept (concept 3). Assuming this decomposition the larger concept can be described by the following ACL query:

```
@
        Assignment-Stmt
                abs-gen-desc
                        defines : [?p];
        Iterative-Stmt
                (abs-gen-desc
                        uses : [?TrP],
                        defines : [?DistrVal])
                (abs-exp-desc
                        Keywords : [nextMix])
        {
          abs-gen-desc
                     empty
          SOURCE : "concept1"
          *-Stmt
            abs-gen-desc
                        empty;
          SOURCE : "concept2"
          *-Stmt
            abs-gen-desc
                        uses : [TrP]
        }
        SOURCE : "concept3"
@
```

ACL Query describing a large-grain concept

Observe the usage of the *SOURCE* macro as well as the use of don't care statements (i.e. **-Statement*) as gluing material between the fine-grain concepts. Discrete description of each smaller plan exist in the files included by the macros. The contents of these files (i.e. concept1-3) are presented below.

File : Concept1

```
@
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc empty
        Then
        {
          abs-gen-desc
                     empty
          Assignment-Stmt
                abs-gen-desc
                        uses :  [idx],
                        defines : [?DistrVal];
          *-Stmt
                abs-gen-desc
                        defines : [idx,distTested]
        }
@
```

ACL Query describing first sub concept

File : Concept2

```
@
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc
                        keywords : [contribution,TrP,DistrVal,idx]
        Then
          Assignment-Stmt
                abs-gen-desc
                             uses : [contribution],
                             defines : [?p]
@
```

ACL Query describing second sub concept

File : Concept3

```
@
          Assignment-Stmt
               abs-gen-desc
                     defines : [mixTested];
          Assignment-Stmt
               abs-gen-desc
                     uses : [?p],
                     defines : [mix,Value]
@
```

ACL Query describing code segments in lines 11 & 12

The reader must notice that the use of only one bind variable in an ACL query does not make sense. However if we combine the three queries we notice that there are no single bind variables.

Using the first concept only as input to our system we found 91 occurrences of it in the code. The second sub-concept appears only 3 times in the Recognizer's code where as the third concept occurs only twice. Using the generic query we managed to successfully locate the concept in question in the Recognizer's code.

This method of hierarchical plan recognition can be adopted to describe and identify large-grain concepts in the code when smaller sub-concepts have been identified.

5.4 Testing results presentation and analysis

We believe that the framework introduced in this work can be used for information retrieval in general and not only in the design recovery process. The focus of our testing was to estimate how effective a system using this new framework is. Our secondary objective was to explore the process of creating a good concept description using the query language we introduced. During result analysis we will report our conclusions on the later subject.

The most widely used measures for evaluating retrieval effectiveness are *Recall* and *Precision* [59]. Recall is defined as the proportion of relevant material;i.e. it measure how well the considered system retrieves all the relevant components. Precision is defined as the proportion of retrieved material which is relevant; i.e., it measures how well the system retrieves only the relevant components. Recall can also be interpreted as the probability that a relevant component will be retrieved, and precision as the probability that a retrieved component will be relevant [5].

Recall and precision can be defined more formally as follows. Let C be the universe of possible retrieved elements, for a design recovery system this would be the set of all design plans - concepts in a system. For each query, C can be partitioned into two disjoint sets, R, the set of relevant material, and $\bar{R}$, the set of irrelevant material. The information retrieval system will then retrieve a set of components c than can also be partitioned into relevant and irrelevant material, r and $\bar{r}$ respectively. Recall and precision are then defined as :

$$Recall = \frac{r}{R}$$

$$Precision = \frac{r}{C}$$

It is obvious that recall and precision measurement require the ability to distinguish between relevant and irrelevant material. Relevance judgements are always debatable. In our case the most difficult task was to find all possible relevant material in our input; i.e. recognizing all occurrences of a concept in a program so we could accurately measure recall. For the smaller subject systems this tedious task was possible but for the larger ones we had to rely on the system's modularity.

In order to produce meaningful diagrams we also had to quantify in some way the expressiveness of each ACL query; to achieve this we adopted a simple formula to calculate a weight for each query. The weight should be higher for precise queries and lower for abstract ones. As expected testing indicates that using exact statements instead of don't care or abstract statements (e.g. the *Iterative Statement*) to describe a specific statement yields higher precision results. Increased precision was observed as well when non abstract features are used to describe the features of a code segment. Use of abstract features, in the form of bind variables, results in lower precision for well formed queries. These observations led us to create the following simple weight formula:

$$Weight = \#Statements \cdot Cost1 + \#Non\_Abstract\_Features \cdot Cost2 +$$

$$\#Abstract\_Features \cdot Cost3 \qquad (I)$$

Where Cost1 = 3, Cost2 = 2 and Cost3 = 1.

For every subject system we located five concepts and for each concept we came up with six to ten different descriptions. These descriptions were formed by varying the:

- number of abstract and non abstract features.
- number and type of statements in the query.

As a result these thirty concepts were expressed in two hundred and five different queries. Queries describing the same concept mainly differ in weight which indicates the degree of abstractness and expressiveness. The series of diagrams that follows focuses more on qualitative results rather than quantitative ones. It would be easy to come up with several different ways of expressing a concept each one yielding a different precision. However our major interest and objective was to capture the general behavior of the system when certain parameters change. For this reason diagrams are expressed across several different queries describing different concepts for each subject system.

The first set of diagrams presented in figure 5.1 presents the relation between precision and the average similarity measure reported for each query for a particular subject system. As expected we observe an analogy between these two quantities. As our queries become more precise the average similarity measure reported also increases and viceversa. The average similarity measure is calculated as the average of the similarity measure calculated for each reported instance of a concept described by an ACL query. It is interesting to notice that some times precision remains constant when small differences in the average similarity measure occur: the reason for this is that adding more features to our description after a certain point does not have a significant effect on precision but will most certainly change the average similarity measure reported.

Additional conclusions regarding the quality and effectiveness of a query in ACL can be drawn from our next diagram set (see figures 5.2,5.3). These diagrams show the relation among the query precision, the number of retrieved concept instantiations in a subject system and two main factors of the weight formula which also reflect the expressiveness of the query, namely the number of abstract and non abstract features specified in the ACL query.

These diagrams show some of the characteristics of the system. The general rule is that the more abstract a query is made the bigger the number of retrieved concept instantiations and the lower the precision would be. Using ACL for C there is a number of ways an analyst can make a query more precise, namely the analyst can:

1. use specific statements instead of generic or don't care statements to describe a particular code segment,
2. utilize more non abstract features to describe properties of a code segment and lastly
3. avoid the use of abstract features (i.e. bind variables) as much as possible.

The following points can be verified by examining the diagrams in figures 5.2 and 5.3. We see that increasing the number of abstract statements and non abstract features results in less concept instantiations reported and better precision. On the

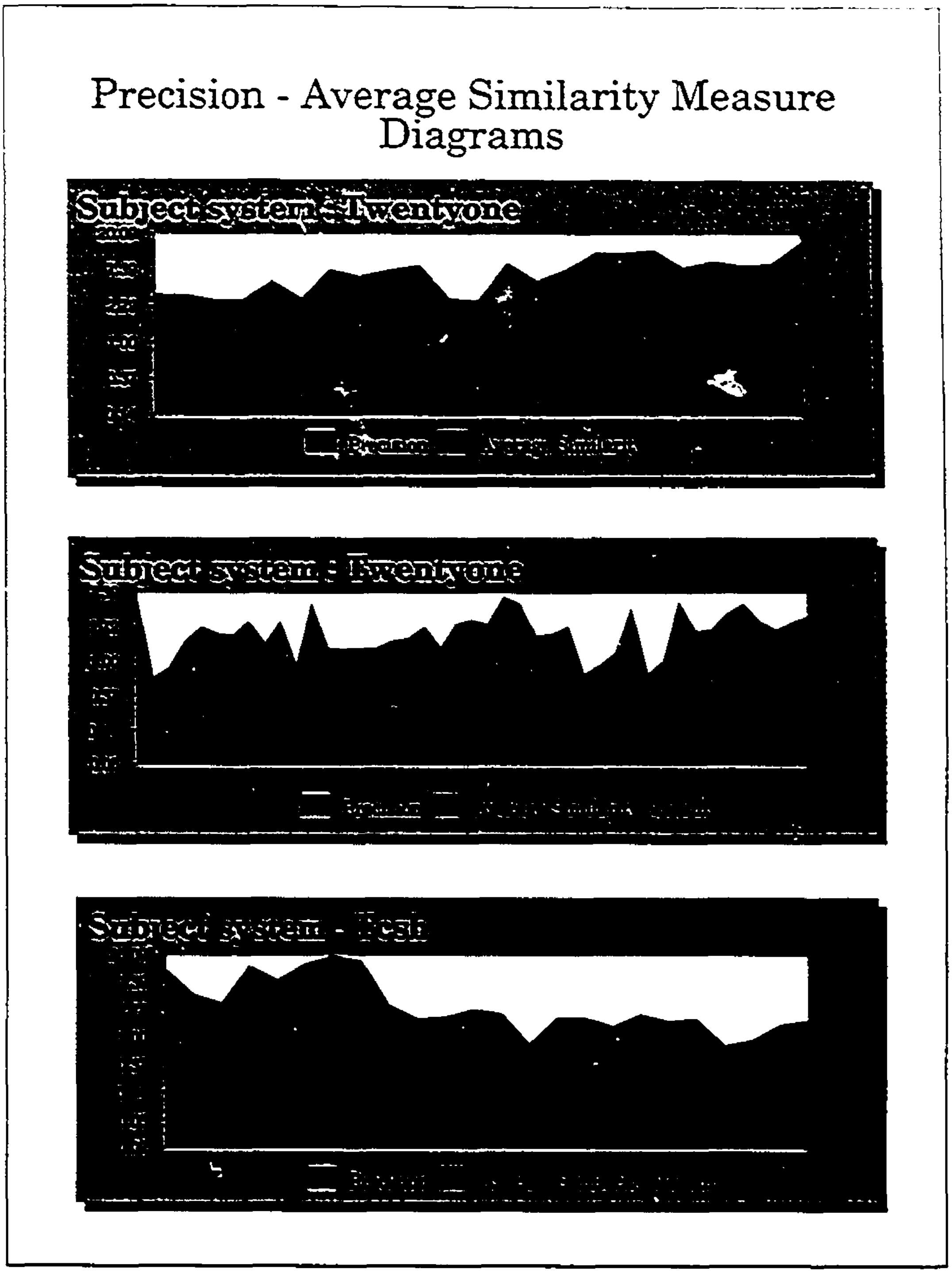


Figure 5.1: Precision - Average Similarity Measure Diagrams.

other hand increasing the number of abstract features leads to an increase of reported concept instantiations and lower precision.

Another interesting dependency can be observed in the third set of diagrams shown in figures 5.4 and 5.5. The way we defined the weight for an ACL query, a greater weight corresponds to more precise thus less abstract queries (see formula 1). This relation is shown in the diagrams. For a well designed query increasing weight should result in greater precision.

Finally our last set of diagrams shows the existing relation between *Recall* and *Precision* (figure 5.6). As expected these two quantities are dependent and asymmetrical. The higher the *Precision* achieved by a query the lower the *Recall* will be. Making a query more abstract means that we specify less features and use the query language in a less restrictive way. Inevitably less logical constraints will result in more irrelevant components retrieved and lower precision. Partial match allows to virtually retrieve all the relevant info but usually this comes with a price in precision.

Summarizing our results we can say that:

- precision is highly correlated with the similarity measure. This result verifies the correctness and effectiveness of the comparison algorithm used,
- the number of features (abstract and non abstract) is highly correlated with the number of concepts retrieved,
- increasing the number of abstract statements, above a certain "threshold", in the query seems not to affect significantly the number of retrieved concepts,
- queries using only abstract features yield noisy results and consequently high recall values,
- effective queries have to use both abstract and non abstract features in a balanced number and specific statements rather than abstract statements,
- when the precision drops the recall increases. This means that more abstract queries that introduce more noisy results (lower precision) tend to capture more instances of a concept in the system (higher recall). Moreover recall is relatively stable for most queries and that verifies the completeness of the features selected.

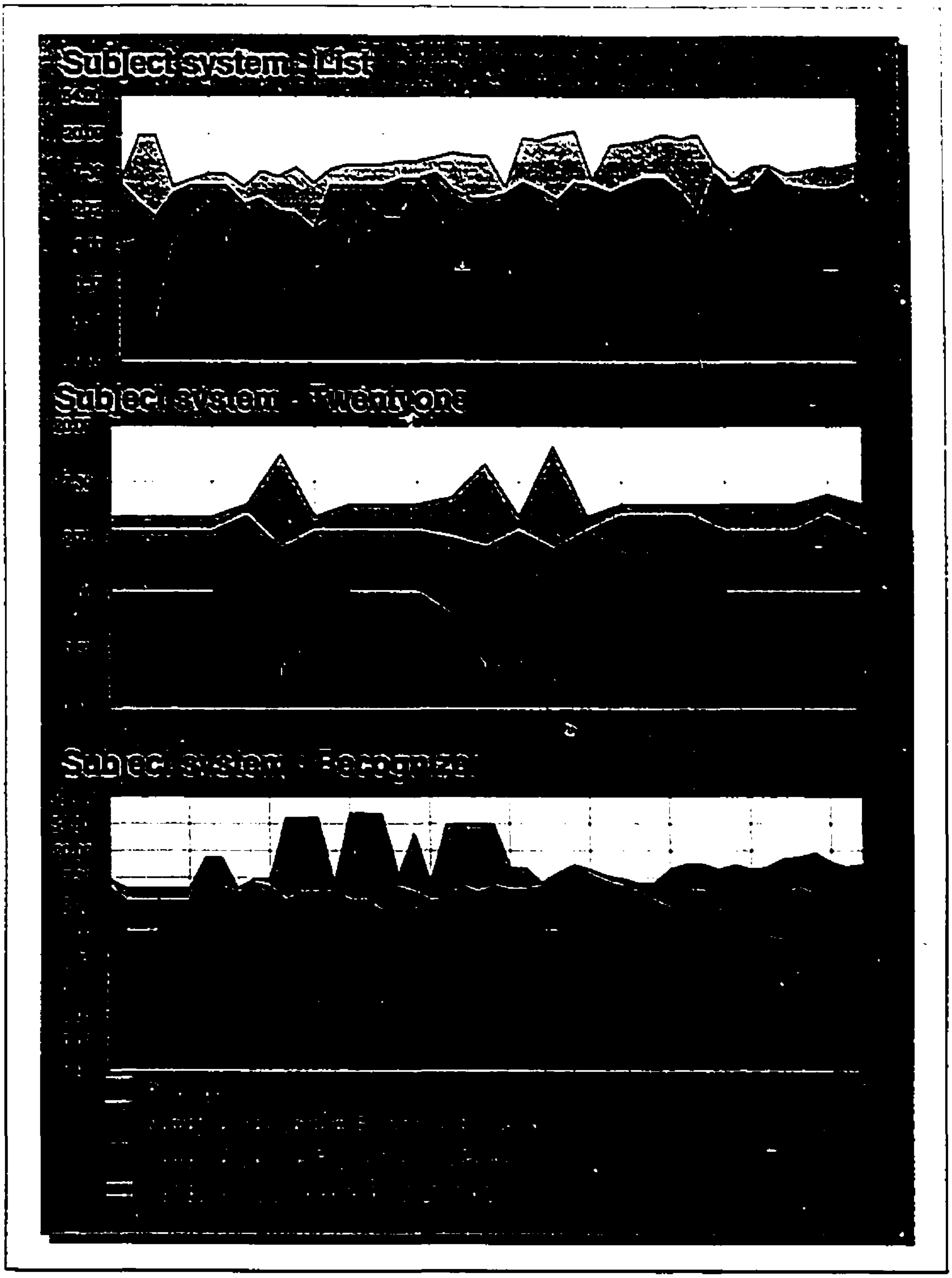


Figure 5.2: Retrieved Concept Instantiations - Weight Factors Diagrams.

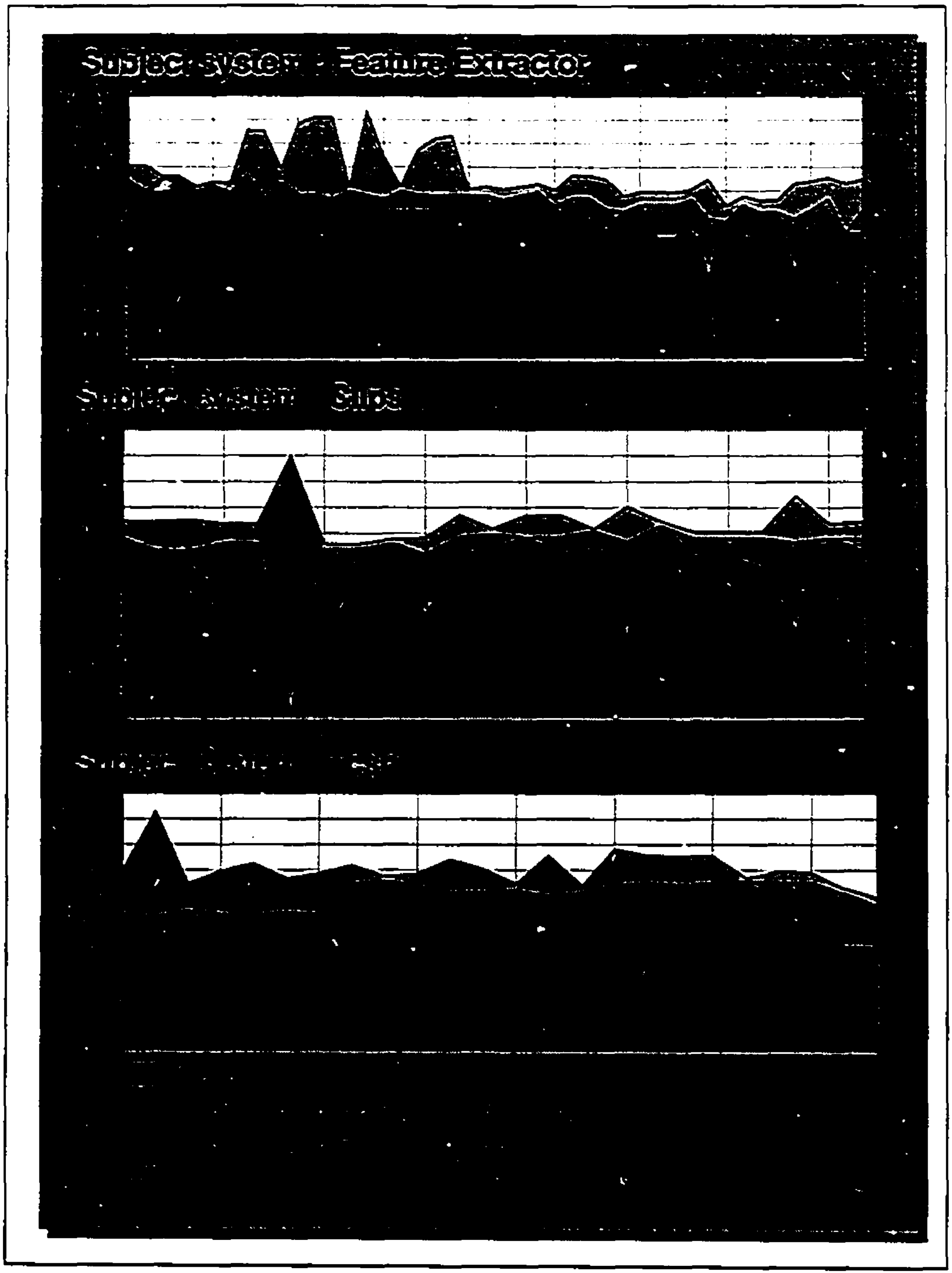
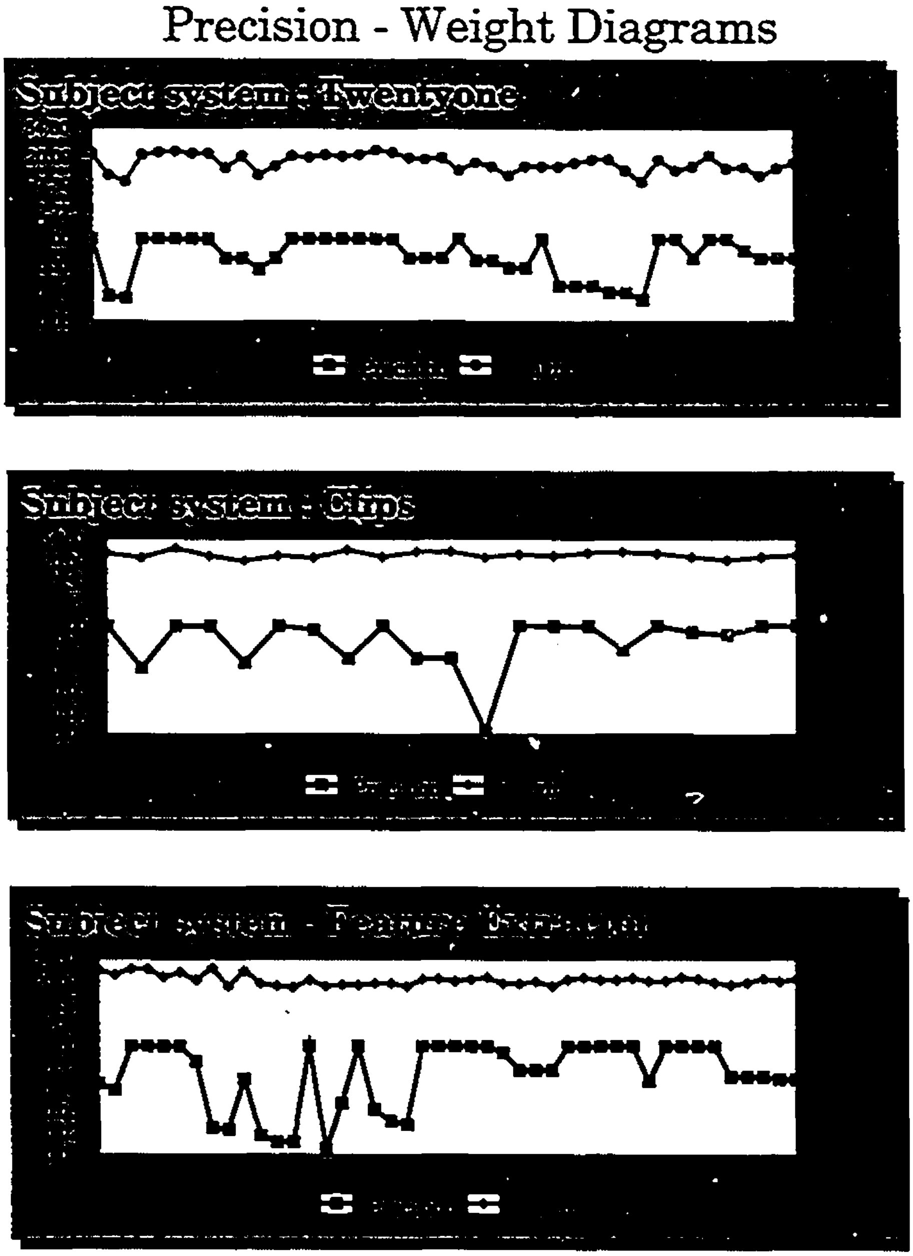


Figure 5.3: Retrieved Concept Instantiations - Weight Factors Diagrams.



Precision - Weight Diagrams

Subject system : Twentyone

Subject system : Clips

Subject system - Feature [illegible]

Figure 5.4: Precision - Query Weight Diagrams.

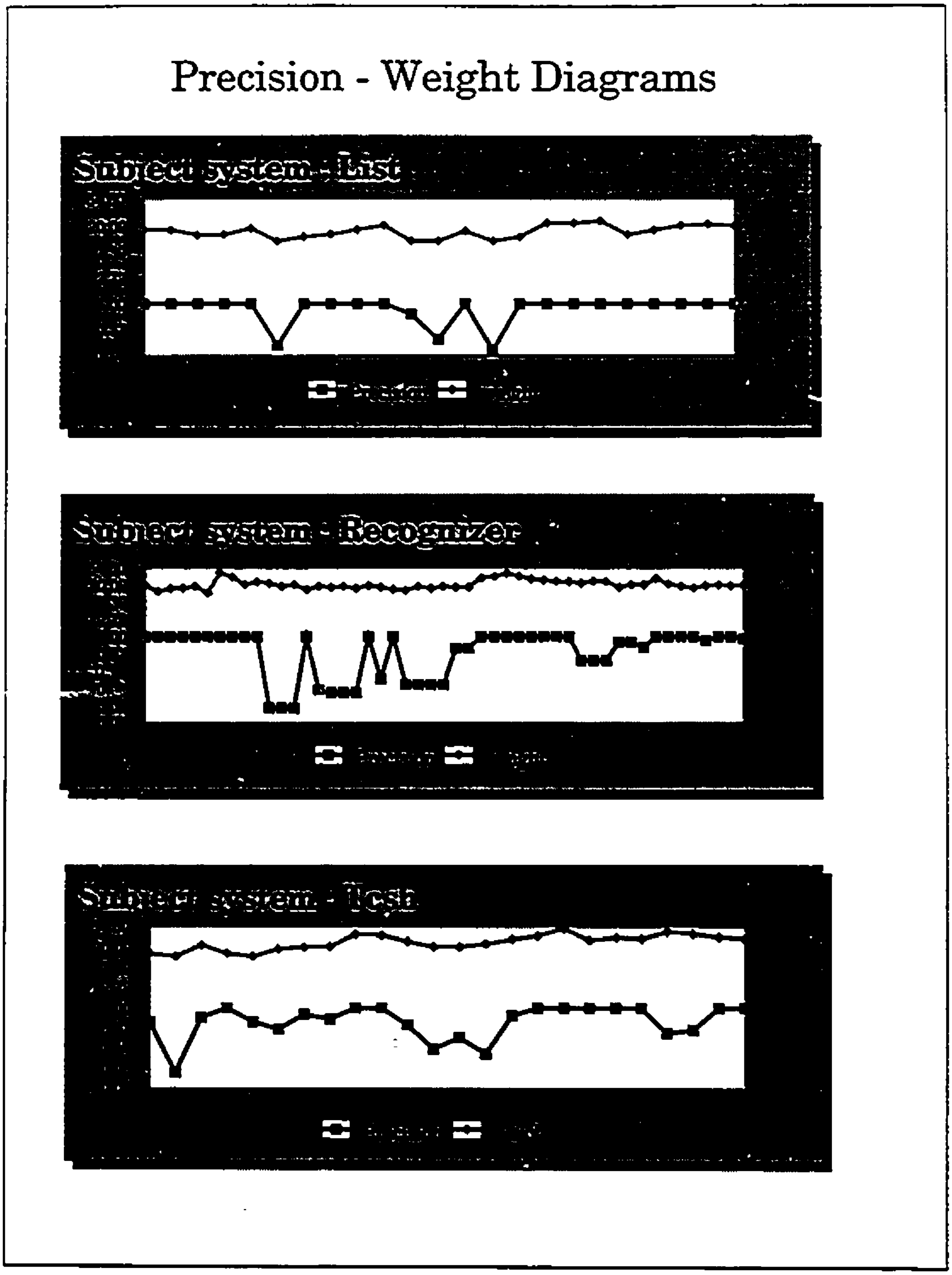


Figure 5.5: Precision - Query Weight Diagrams.

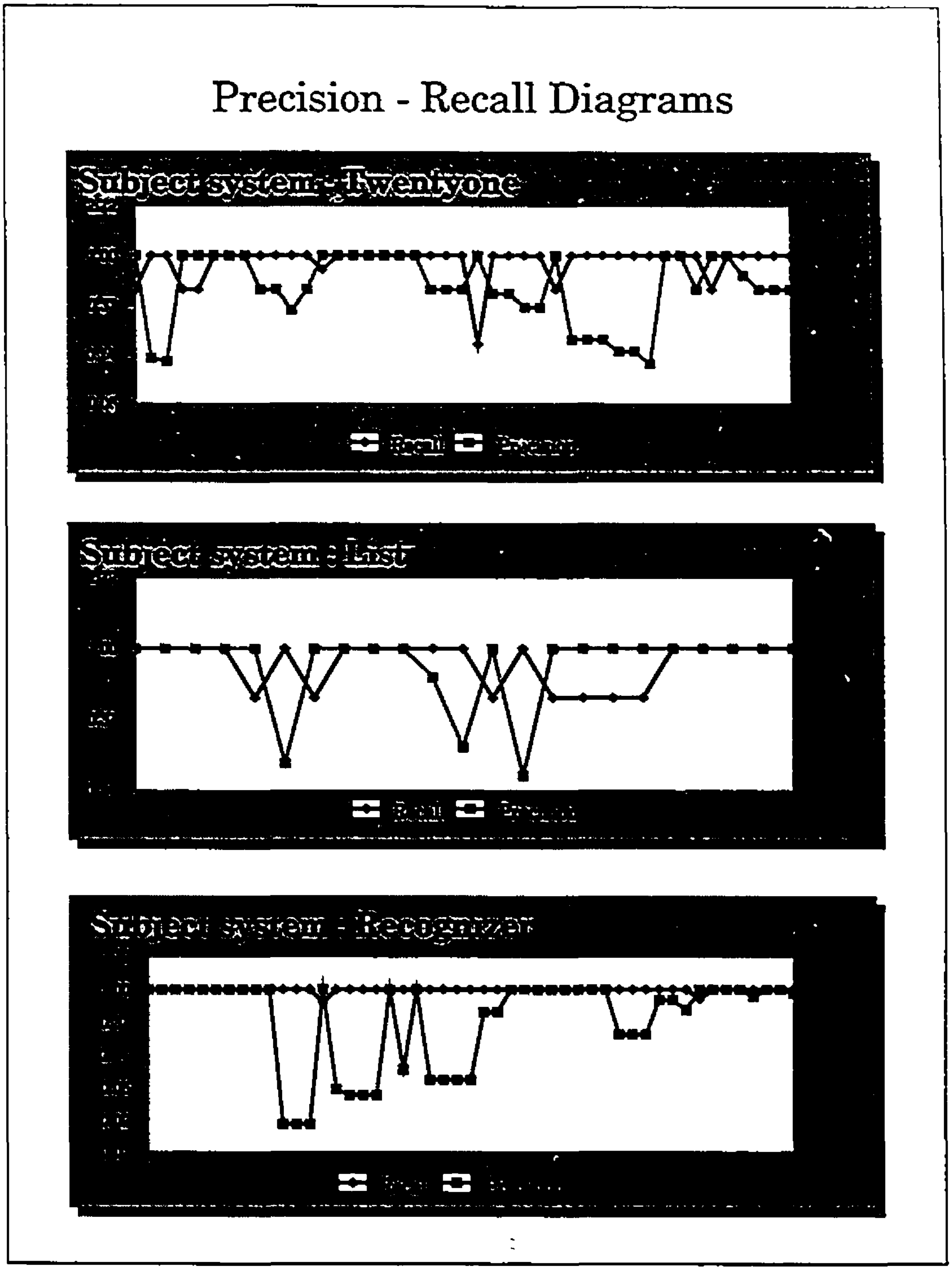
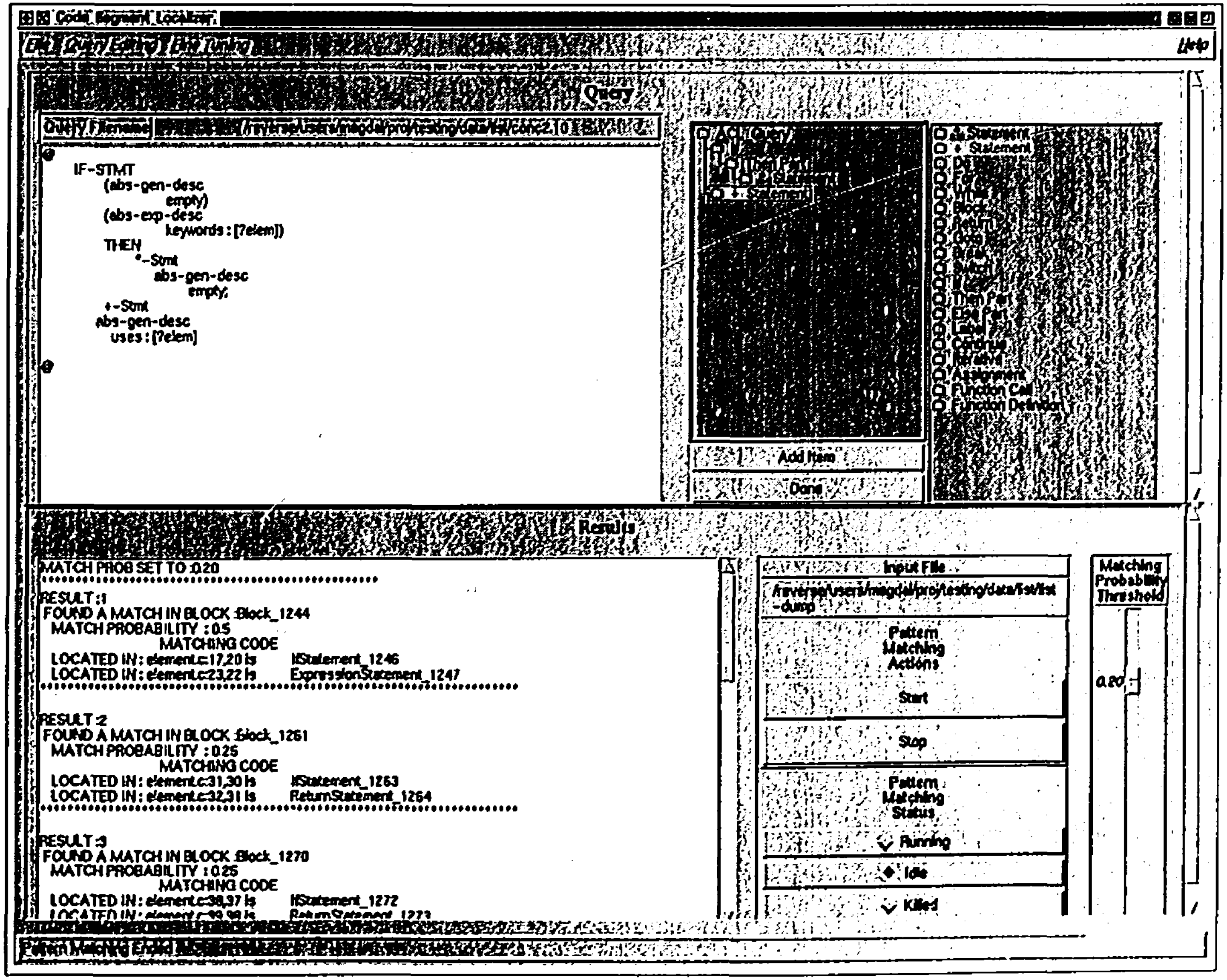


Figure 5.6: Precision - Recall Diagrams.

Finally new users can easily improve their tool usage skills by taking advantage of the graphical interface and the query editor supplied. A screen dump of the interface is shown in figure 5.7.



```
IF-STMT
     (abs-gen-desc
            empty)
     (abs-exp-desc
            keywords : [?elem])
     THEN
          *-Stmt
            abs-gen-desc
                 empty;
       +-Stmt
     abs-gen-desc
       uses : [?elem]
```

```
MATCH PROB SET TO :0.20
****************************************
RESULT :1
 FOUND A MATCH IN BLOCK :Block_1244
  MATCH PROBABILITY :0.5
                  MATCHING CODE
  LOCATED IN : element.c:17,20 ls        IfStatement_1246
  LOCATED IN : element.c:23,22 ls        ExpressionStatement_1247
************************************************************

RESULT :2
 FOUND A MATCH IN BLOCK :Block_1261
  MATCH PROBABILITY :0.25
                  MATCHING CODE
  LOCATED IN : element.c:31,30 ls        IfStatement_1263
  LOCATED IN : element.c:32,31 ls        ReturnStatement_1264
************************************************************

RESULT :3
 FOUND A MATCH IN BLOCK :Block_1270
  MATCH PROBABILITY :0.25
                  MATCHING CODE
  LOCATED IN : element.c:38,37 ls        IfStatement_1272
```

Input File: /reverse/users/magdal/proj/testing/data/list/list-dump

Figure 5.7: The Graphical User Interface.

Chapter 6

Conclusions

In previous chapters we presented the framework created and adopted for our system as well as the experimental results we obtained using the system to locate concepts in various C programs. This chapter discusses possible directions for future research in order to improve the system and presents a summary of our conclusions.

6.1 Future work

There are two major directions for future improvement of both the framework and the system we described. The first is enhancing the capabilities of the system and the second is extending its scope. The following sections explore these directions.

System enhancement

Significant performance improvement is possible by introducing parallelism in the algorithm. The nature of the algorithm makes it an ideal candidate for parallelization. To be more specific what we suggest here is parallelizing the matching process after the candidate starting points have been computed. Knowing how many distinct cases have to be considered we can then "fork" as many processes to handle each case concurrently. This is of course a non trivial task. One must consider possible overhead and space requirements due to massive copying of structures that is going to occur. Further more we must estimate the effect of using parallelism on our framework's complexity. Several techniques for parallel programming in C++ are being proposed and we are currently going through the literature to estimate the effort needed to accomplish this task.

Another possible improvement would be the implementation of several low level feature comparison methods. The analyst would then have the opportunity to choose the one he finds more suitable depending on the task the system has to accomplish. In the current implementation feature comparison is done using exact string comparison and metric distance is calculated using the euclidean distance. One possibility would be to calculate lexicographic distances [26] between code features and their descriptions in the ACL query. We could also use a different formula to compute metric distances. The analyst would have the opportunity to choose the desired method of feature comparison from a list of available methods in the graphical interface and fine tune it by changing certain parameters or thresholds. For example in the current version the user can adjust the threshold used to characterize a recognized statement as a possible match as well as the threshold used to check the metrics distance of two code segments. If we use lexicographic distances for feature comparison the user should be able to specify the minimum number of characters a feature should have so that the comparison is meaningful.

In the prototype, implemented for the REVENGE project, the Abstract Concept Language (ACL) is more powerful. The analyst can specify the type of a variable in the query and use logical operators to define the sequence of abstract statements. Those features were not included in our version of the system mainly due to time constraints. We estimate that existence of these features is also a possible enhancement. The existence of logical operators can be particularly useful in order to solve some interesting problems that arise from the possible implementation diversity of a concept. In every programming for example, the programmer can sometimes interchange two statements that are not dependent on each other without changing the functionality of the code segment containing these statements. To capture these cases we could use the logical OR operator in ACL and describe our pattern as : $A_i||A_j$, this would result in the creation of two sequential models $A1; A2$ and $A2; A1$. The one that maximizes the overall matching probability calculated would be chosen. In our present implementation the only way to solve this problem is to use don't care statements (i.e. the **-Statement* or the *+-Statement*).

A useful enhancement would be to graphically present our results. We are currently exploring ways of representing graphically the AST and the matching results. The AST will be represented as a simple n-ary tree. Nodes in the tree correspond to nodes in the AST and thus to statements in the original code. The analyst will have the ability to click on any node and get information about the node's features. Recognized concepts can then be presented to the analyst as highlighted areas (set of nodes) in the tree. Implementing this GUI enhancement is an interesting task. A new extension to Tcl/Tk exists that allows the display of dynamically created trees. We estimate that presenting the whole AST can be time consuming, however it would be possible for the analyst to choose between displaying the whole AST, just the parts of the AST that contain recognized results or only some preselected parts of it.

The design presented in chapter four is the result of several iterations over our initial design ideas. Introducing new features to our system will inevitably lead to further evolvement of the design.

System Extension

An important step toward the evolution of the system design would also be the use of the framework for a new target language. Possible target languages can be HTML, Pascal or simply "structured" text. At the moment we find HTML and "structured" text the most interesting candidates mainly because using them would help us to further evaluate the system from the information retrieval point of view.

Extending the tool with a new target language is a three step process. First we need to create a domain model for the new language able to capture the language's basic constructs and their main features. For HTML pages, paragraphs, sentences, applets and images can be considered basic constructs. Each basic construct has particular features and also shares some common features with other classes. Links and references, maps or background and foreground information can be considered features of an HTML document. The next step is the creation of a parser for the language. This parser should produce an intermediate representation of the "source code" in the form of an AST. Nodes of the AST would be objects of the classes specified in the domain model of the language. Finally we need to implement meaningful feature comparison functions for the language. By plugging the newly created elements to the existing framework we can then use our main code segment localization algorithm to locate occurrence of a "code" segment in the input.

In terms of effort needed to accomplish these steps we have been able to confirm that the creation of the domain model is the most time consuming and challenging step. For most programming languages publically available parsers exist. We found particularly useful to have such a parser in our initial resources. Going through the parser we can factorize entities and create primary abstractions that can subsequently drive the creation of the domain model. We already have a parser for "structured" text, which is plain text with some simple tags to indicate end of paragraphs or pages. HTML parsers are available and are also considered at the moment as possible starting points.

Finally it would be useful to incorporate in the system a small knowledge base where we could store the recognized concepts and create small libraries of plans for each subject system we examine. The global repository, currently used as a source for our input, is a possible candidate. Using the domain model of the target language we can form s-expressions describing a concept and store them in the global repository. Using the global repository will permit the sharing of concept descriptions among the participating tools in the cooperative environment.

6.2 Summary of conclusions

The purpose of the work reported in this document was the creation and use of a generalized framework for information retrieval on large spaces containing structured data. The particular implementation is applied to the program understanding domain.

The framework introduced was used to create a code segment localizer which can be used for concept localization in C programs. In the heart of this framework is an algorithm that performs information retrieval based on complete or partial matching of structured features. Concept detection and localization is a crucial part of the design recovery process which, in turn, constitutes a vital task of the maintenance process. The resulting code segment localizer can be part of a larger cooperative environment of CASE tools created for the REVENGE project. The main components of the framework are:

- a flexible and simplified domain model of the target language,
- parsing facilities for conversion to an intermediate representation (AST) of both the "source code" and the query describing the concept and
- a comparison engine implementing the main localization algorithm using the Viterbi dynamic programming algorithm and Markov Models.

An object oriented approach, and programming language (namely C++), was chosen for the implementation of the framework in order to achieve greater modularity, extensibility and ease of maintenance. After several iterations of introducing enhancements to the system allowed us to conclude that extensibility and maintainability were achieved.

Extensive testing proved the capabilities of our framework and provided satisfactory results for a large range of subject systems and concepts.

We strongly believe that the generic framework presented in this report can be used to perform information retrieval in a variety of fields as long as information in the search space presents some structure and is described using formal, structure oriented patterns of features.

Appendix A

The Abstract Concept Language grammar

In this appendix we present the grammar of the Abstract Query Language we used for C programs in Backus Normal Form. Reserved words of the language appear in bold (a complete table for reserved words appear at the end), C-like syntax is used for comments.

```
<query>            : ATSIGN <stmt_states> ATSIGN
<stmt_states>      : /* empty */
                   | <stmt_state> <stmt_states>
<stmt_state>       : SEMICOLON <stmt_descr>
                   | <stmt_descr>
<stmt_descr>       : <if_stmt>
                   | <include_plan>
                   | <iter_stmt>
                   | <while_stmt>
                   | <do_stmt>
                   | <for_stmt>
                   | <ret_stmt>
                   | <goto_stmt>
                   | <cont_stmt>
                   | <break_stmt>
                   | <switch_stmt>
                   | <label_stmt>
                   | <assign_stmt>
                   | <fnccall_stmt>
                   | <block_stmt>
```

```
                        | <zero_or_more_stmt>
                        | <one_or_more_stmt>
                        | <function_def>
                        | <expr_stmt>
<include_plan>          : SOURCE <STRING>
<if_stmt>               : IFSTMT <gen_descr> <cond_descr> THEN <stmt_descr>
                           ELSE <stmt_descr>
                        | IFSTMT <gen_descr> <cond_descr> THEN <stmt_descr>
<iter_stmt>             : ITERSTMT <gen_descr> <cond_descr> <stmt_descr>
<while_stmt>            : WHILESTMT <gen_descr> <cond_descr> <stmt_descr>
<do_stmt>               : DOSTMT <gen_descr> <cond_descr> <stmt_descr>
<for_stmt>              : FORSTMT <gen_descr> LPAREN <pattern_descr>
                          SEMICOLON <pattern_descr> SEMICOLON<pattern_descr>
                          RPAREN <stmt_descr>
<ret_stmt>              : RETSTMT <gen_descr>
<goto_stmt>             : GOTOSTMT <gen_descr>
<cont_stmt>             : CONTSTMT
<break_stmt>            : BREAKSTMT
<switch_stmt>           : SWITCHSTMT <gen_descr> <cond_descr> <stmt_descr>
<label_stmt>            : LABELSTMT <gen_descr>
<assign_stmt>           : ASSIGNSTMT <gen_descr>
<fnccall_stmt>          : FNCCALLSTMT IDENTIFIER <gen_descr>
<block_stmt>            : LCBRACKET <gen_descr> <stmt_states> RCBRACKET
<zero_or_more_stmt>     : ZEROMORESTMT <gen_descr>
<one_or_more_stmt>      : ONEMORESTMT <gen_descr>
<function_def>          : FUNCTION IDENTIFIER <gen_descr> block_stmt
<expr_stmt>             : EXPRSTMT <gen_descr> block_stmt
<gen_descr>             : LPAREN <gen_pattern_descr> RPAREN
                        | <gen_pattern_descr>
<gen_pattern_descr>     : ABSGENDESCR <pattern_descr>1
<cond_descr>            : LPAREN <pattern_descr> RPAREN
                        | <pattern_descr>
<pattern_descr>         : ABSEXPRDESCR <pattern_descr>1
<pattern_descr1>        : EMPTY
                        | <features_descr>
<features_descr>        : <uses_descr> <defines_descr> <keywords_descr> <metrics_descr>
<uses_descr>            :/* empty */
                        | USES LBRACKET <identifier_seq> RBRACKET
                        | USES LBRACKET <identifier_seq> RBRACKET COMMA
<defines_descr>         :/* empty */
                        | DEFINES LBRACKET <identifier_seq> RBRACKET
```

```
                      | DEFINES LBRACKET <identifier_seq> RBRACKET
                      COMMA
<keywords_descr>      :/* empty */
                      | KEYWORDS LBRACKET <identifier_seq> RBRACKET
                      | KEYWORDS LBRACKET <identifier_seq> RBRACKET
                        COMMA
<metrics_descr>       :/* empty */
                      | METRICS LBRACKET <Float> COMMA <Float>
                        COMMA <Float> COMMA <Float> COMMA <Float>
                        RBRACKET
<identifier_seq>      : <identifier_seq1>
<identifier_seq1>     : <identifier_seq2> IDENTIFIER
                      | <identifier_seq2> QUESTION IDENTIFIER
<identifier_seq2>     : /* empty */
                      | <identifier_seq2> IDENTIFIER COMMA
                      | <identifier_seq2> QUESTION IDENTIFIER COMMA
<Float>               : FLOAT
```

Reserved Word Symbol	*Actual Reserved Word*
ABSEXPRDESCR	abs-exp-desc Abs-Exp-Desc ABS-EXP-DESC
ABSGENDESCR	abs-gen-desc Abs-Gen-Desc ABS-GEN-DESC
FUNCTION	Function-Def function-def FUNCTION-DEF

IFSTMT	if-stmt
	If-Stmt
	IF-STMT

THEN	then Then THEN

ELSE	else
	Else
	ELSE

ITERSTMT	iterative-stmt Iterative-Stmt ITERATIVE-STMT
WHILESTMT	while-stmt While-Stmt WHILE-STMT
DOSTMT	do-stmt Do-Stmt DO-STMT
FORSTMT	for-stmt For-Stmt FOR-STMT

RETSTMT	return-stmt
	Return-Stmt
	RETURN-STMT

GOTOSTMT	goto-stmt Goto-Stmt GOTO-STMT
EXPRSTMT	expr-stmt Expr-Stmt EXPR-STMT

Table A.1: ACL's Reserved Words [I]

Reserved Word Symbol	*Actual Reserved Word*
CONTSTMT	continue Continue CONTINUE
BREAKSTMT	break Break BREAK
SWITCHSTMT	switch-stmt Switch-Stmt SWITCH-STMT
LABELSTMT	labelled-Stmt Labelled-Stmt LABELLED-STMT
ASSIGNSTMT	assignment-stmt Assignment-Stmt ASSIGNMENT-STMT
FNCCALLSTMT	function-call Function-Call FUNCTION-CALL
ZEROMORESTMT	*-stmt *-Stmt *-STMT
ONEMORESTMT	+-stmt +-Stmt +-STMT
EMPTY	empty Empty EMPTY
KEYWORDS	keywords : Keywords : KEYWORDS :
DEFINES	defines : Defines : DEFINES :
USES	uses : Uses : USES :

Table A.2: ACL's Reserved Words [II]

Reserved Word Symbol	*Actual Reserved Word*
METRICS	metrics : Metrics : METRICS :
SOURCE	source : Source : SOURCE :
LBRACKET	[
RBRACKET	]
RBRACKET	(
RPAREN	)
LCBRACKET	{
RCBRACKET	}
ATSIGN	@
COMMA	,
SEMICOLON	;
COLON	:
QUESTION	?

Table A.3: ACL's Reserved Words [III]

Appendix B

Examples of concepts

Subject System : *Twentyone*

Concept description

For each player check if he/she placed a bet and if so then deal a new card and update the necessary variables.

```
@

     Iterative-Stmt
                (abs-gen-desc empty)
                (abs-exp-desc
                        keywords : [?player])
        {
          abs-gen-desc empty
                +-Stmt
                        abs-gen-desc
                                uses : [?player],
                                defines : [?card];
                +-Stmt
                        abs-gen-desc
                                empty;
                 Assignment-Stmt
                        abs-gen-desc
                                uses : [?card];
                +-Stmt
                        abs-gen-desc empty
        }
@
```

Example of reported concept instantiation

```
for ( player = 0 ; player < num_players ; ++player )
 { card = players [ player ] . bet ? deal_card ( ) : ' ' ;
  ( void ) printf ( "\t%c" , card ) ;
  players [ player ] . cards [ players [ player ] . num_cards++ ] =
   card ;
  players [ player ] . cards [ players [ player ] . num_cards ] = '\0' ;
  players [ player ] . busted = 0 ;
  players [ player ] . split = 0 ;
 }
```

Subject System : *List*

Concept description

Check if memory allocation for an element has failed and initialize element's fields.

```
@
    If-Stmt
        abs-gen-desc empty
        abs-exp-desc
            keywords : [elem]
    Then
    {
     abs-gen-desc
          empty
    *-Stmt
      abs-gen-desc
              empty
    };

    *-Stmt
      abs-gen-desc
              uses : [elem]
@
```

Example of reported concept instantiation

```
if (elem == NULL)
{
    fprintf(stderr,"elementcreate: malloc failed, out of memory???\n");
    return NULL;
}

elem->next = NULL;
elem->info = i;
```

Subject System : *Recognizer*

Concept description

Part of the transition probability calculation. For each active transition check if probability is already calculated if not calculate it; then check if this newly calculated probability is bigger than the max probability so far and if so update the current maximum. Finally perform some simple initializations.

```
@
        Iterative-Stmt
                (abs-gen-desc
                        uses : [?TrP],
                        defines : [?DistrVal])
                (abs-exp-desc
                        empty)
        {
          abs-gen-desc
                     empty
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc empty
        Then
        {
          abs-gen-desc
                     empty
          Assignment-Stmt
                abs-gen-desc
                        defines : [?DistrVal];
        *-Stmt
            abs-gen-desc
                          empty
        };
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc empty
        Then
          Assignment-Stmt
                abs-gen-desc
                          empty;
        *-Stmt
            abs-gen-desc
                          uses : [?TrP]
        }
@
```

Example of reported concept instantiation

```
do {
    if (!(distTested[idx = TrP->DistrIdx])) {
      DistrVal[idx] = EvalDistr(&DistrList[idx].obs);
      distTested[idx] = TRUE;
     }
    if(p<(contribution=TrP->Prob+DistrVal[idx]))
       p = contribution;
    TrP++;
   } while(++i < *nextMix);
   *mixTested = TRUE;
   mix->Value = p;
```

Subject System : *Feature Extractor*

Concept description

Check the energy level and if is less than the current minimum update the minimum; also if its less than a certain filter value replace the current threshold with this filter value.

```
@
      Assignment-Stmt
                abs-gen-desc
                        uses : [?enertmp];
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc empty
        Then
        Assignment-Stmt
                abs-gen-desc
                        defines : [?enertmp];
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc
                        keywords : [?enertmp]
        Then
        {
          abs-gen-desc
                     empty

          *-Stmt
                abs-gen-desc
                        empty
        }
@
```

Example of reported concept instantiation

```
enertmp /= MelWeight[j];
if (enertmp < min_energy) enertmp = min_energy;
if (enertmp < SilFilt[j]  ) {
    fprintf(stderr,"REPLACING (1)  %f with threshold %f\n",enertmp,SilFilt[j]);
    FiltEnergy[j] = SilFilt[j];
}
```

Subject System : *CLIPS*

Concept description

Check the value of a pointer and if it is NULL then adjust the menu and code variables.

```
@
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc
                        keywords : [eptr]
        Then
        {
          abs-gen-desc
                     empty
       *-Stmt
            abs-gen-desc
                             empty
        If-Stmt
                abs-gen-desc empty
                abs-exp-desc
                        keywords : [lptr]
        Then
        {
          abs-gen-desc
                     empty
        *-Stmt
           abs-gen-desc
                   uses : [lptr]
        };

        *-Stmt
```

```
            abs-gen-desc
                             empty
        }
@
```

Example of reported concept instantiation

```
if (eptr == NULL)
  {
   *code = NO_TOPIC;
   if (lptr->curr_menu != NULL)
     {
      *menu = lptr->curr_menu->name;
      return(lptr->curr_menu->offset);
     }
   return(-1);
  }
```

Subject System : *Tcsh*

Concept description

Part of the prompt printing code.

```
@
          Assignment-Stmt
                abs-gen-desc
                     empty;
        Iterative-Stmt
                (abs-gen-desc
                        empty)
                (abs-exp-desc
                        keywords : [wdp,word])
        {
          abs-gen-desc
                     empty
        *-Stmt
            abs-gen-desc
                             empty
          Assignment-Stmt
                abs-gen-desc
                        uses : [wdp,hp],
                        defines : [new,prev,next];
        *-Stmt
            abs-gen-desc
```

```
                        empty;
          Assignment-Stmt
                abs-gen-desc
                        defines : [wdp,next];
        *-Stmt
            abs-gen-desc
                             empty
        };
          Assignment-Stmt
                abs-gen-desc
                            uses : [wdp],
                            defines : [hp,next]
@
```

Example of reported concept instantiation

```
wdp = hp;
 do {
     register struct wordent *new;

     new = (struct wordent *) xmalloc((size_t) sizeof(*wdp));
     new->word = 0;
     new->prev = wdp;
     new->next = hp;
     wdp->next = new;
     wdp = new;
     wdp->word = word();
 } while (wdp->word[0] != '\n');
hp->prev = wdp;
```

Appendix C

A recognition example

A full blown recognition example is presented in the following paragraphs in order to clarify the process presented in chapter 5. Consider the following code segment description :

```
Q1  @
Q2   Assignment-Stmt
Q3      abs-gen-desc
Q4         defines : [Features];
Q5   Iterative-Stmt
Q6     (abs-exp-desc
Q7          keywords : [Control])
Q8   {
Q9    *-Stmt
Q10     abs-gen-desc
Q11             empty
Q12    Assignment-Stmt
Q13      abs-gen-desc
Q14             uses : [Features,CosTable,FiltEnergy]
Q15   }
Q16 @
```

The above query is used to locate an assignment statement that defines a variable called "Features", followed by an iterative statement which uses the keyword-variable

"Control" in its condition. The iterative statement should have a block in its body. Inside the block there should be at least one statement which would be an assignment that uses three variables; namely "Features", "CosTable" and "FiltEnergy". The assignment statement should be the last statement in the block and can be preceded by zero or more other statements.

Given the query described, an APM is formed (see figure C.1). There are some interesting issues in the creation of the APM; both the *Iterative statement* and the *Block statement* are composite objects so sub-APMs are created for each one of them, as a result recognition will be possible through recursive calls of certain functions for each APM.

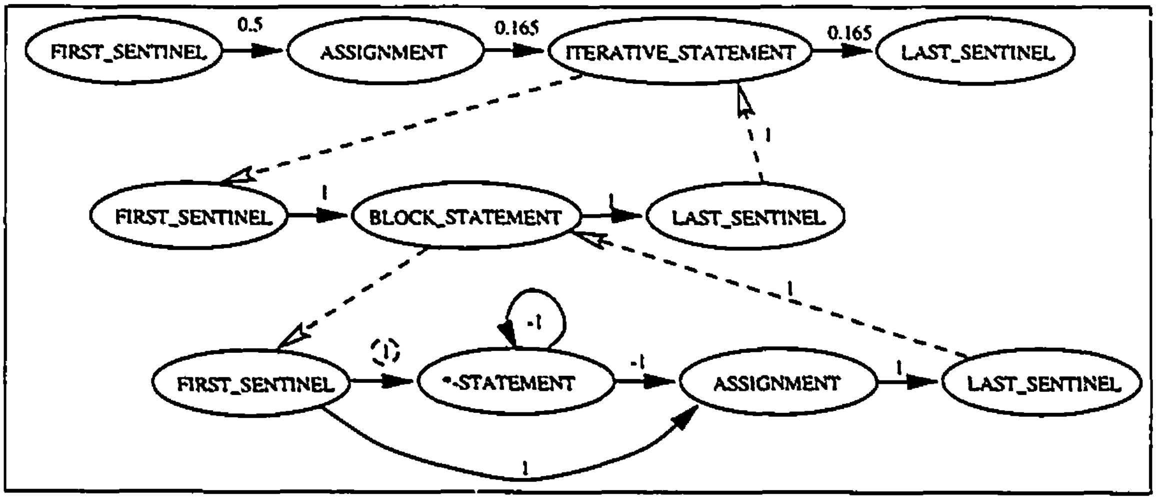


Figure C.1: Resulting APM.

Locating candidate starting points is the initial step of the code segment localization algorithm. Possible starting points for the given query are all the assignments statements in the code. For every possible starting point a call to the *perform_pattern_match* function occurs.

We used this query on the *Feature Extractor* and one possible result was the following piece of code.

```
C1 Features[i + n].PM_mel[j] = 0.0; /* try here */
C2 for (k = 0; k < Control.sa_nfilt; k++)
C3 {
C4  Features[i + n].PM_mel[j] += (CosTable[j][k] * FiltEnergy[k]);
C5  }
```

We will use this piece of code to explain the localization algorithm. The localization process starts by considering the active state in the APM and the active code in the source code AST (T_c). The first state in the APM, the *Assignment State* described in lines Q2 to Q4 in the query, will be compared with the first statement in the AST; line C1 in the code. Initially we perform the three step check to ensure that a comparison is possible. Statement type compatibility, metrics distance and specific features are compared. Type compatibility is successful, metrics and specific features are not checked as they are not specified in the query. A similarity measure

$$P_{COMP}(S_{C1}|A_{Q2})$$

is subsequently computed using formula (6) introduced in chapter 4. Assuming that the statement in the code (line C1) defines two variable names (*Features* and *PM_mel*) then :

$$P_{COMP}(S_{C1}|A_{Q2}) = \frac{1}{v}.\sum_{n=1}^{1} \frac{card(AbstractFeature_{j,n} \cap CodeFeature_{i,n})}{card(AbstractFeature_{j,n} \cup CodeFeature_{i,n})} = \frac{1}{2} = 0.5$$

The value of v is one because only one feature is specified (i.e. variable names defined). To calculate the final probability to be attached to the transition for the *First Sentinel* to the *Assignment Statement* in the APM, the similarity measure calculated is multiplied by the probability of statement type compatibility specified in the SCM. Both statements are of the same simple type (*Assignment*) so this probability is 1. Finally the product is multiplied by the maximum probability in the incoming transitions of the previous state in the APM. The previous state in this case is the *First Sentinel* so this probability is again 1. As a result the overall similarity measure attached to the first transition in the APM is 0.5 . The formula just described is :

$$P_t = P_{COMP}(S_{Ci}|A_{Qj}) \cdot P_{SCM} \cdot P_{MIP} \qquad (7)$$

Where P_t is the transition probability, P_{SCM} is the static probability given by the static model based on statement type criteria and P_{MIP} is the max incoming probability attached in a transition to the previous state in the APM.

The next active state in the APM is the *Iterative Statement* state and the next active state in the code's AST corresponds to the *For Statement* in line C2. Type compatibility, metric distance and specific feature checks are all successful so the calculation of the similarity measure can proceed. The *Iterative Statement* is a composite statement and in order to calculate its total similarity measure we first calculate the similarity measure of its body by calling recursively the *perform_pattern_match* function.

The active states now are: the *Block Statement* state in the APM and the *Block Statement* state in the AST. The three initial checks are again successful and the *Block Statement* being a composite statement causes a second recursive call to the *perform_pattern_match* function in order to calculate the similarity measure for the *Block Statement.*

From the APM we see that possible active states are now both the **_Statement* state and the *Assignment Statement.* The active code is the node in the T_c AST corresponding to the assignment in line C4. Applying the three step check for the **_Statement* and then calculating the similarity measure yields a transition probability equal to one. A transition probability equal to one is calculated for the second active state (i.e. the *Assignment Statement* too assuming that the corresponding node in the T_c AST uses only the variables named "Features", "CosTable", and "FiltEnergy".

The active states in the APM for the next localization step are:

1. the **_Statement* state (previous active state : **_Statement* state),
2. the *Assignment Statement* (previous active state : **_Statement* state) and
3. the *Last Sentinel* state ((previous active state : *Assignment Statement*).

In the code there is no active state, as a result the first two possibilities (i.e. the **_Statement* and *Assignment Statement* fail the initial three step check. On the contrary the last case *Last Sentinel* is successful and the maximum probability from the incoming transitions to the previous active state (i.e. the *Assignment Statement*) is assigned to the transition to the *Last Sentinel* state.

The similarity measure calculated for the body of the *Block Statement* is equal to the transition probability to the *Last Sentinel* and is returned as the result of the recursive call to the *perform_pattern_match* function. Using formula (7) for the *Block Statement* we have :

$$P_t = P_{COMP}(S_{C3}|A_{Q8}) \cdot P_{SCM} \cdot P_{MIP} = 1 \cdot 1 \cdot 1 = 1$$

This transition probability is again assigned to the transition to the *Last Sentinel* state in the second sub-APM and then passed back as the result of the recursive call to the *perform_pattern_match* function for the *Iterative Statement*. Formula (7) for the *Iterative Statement* now looks as follows :

$$P_t = P_{COMP}(S_{C2}|A_{Q5}) \cdot P_{SCM} \cdot P_{MIP} = 1 \cdot 0.33 \cdot 0.5 = 0.165$$

The probability given by the StatiC Model (SCM) is 0.33 (see figure 4.4) and the previous maximum incoming transition probability is the one calculated for the first *Assignment Statement*.

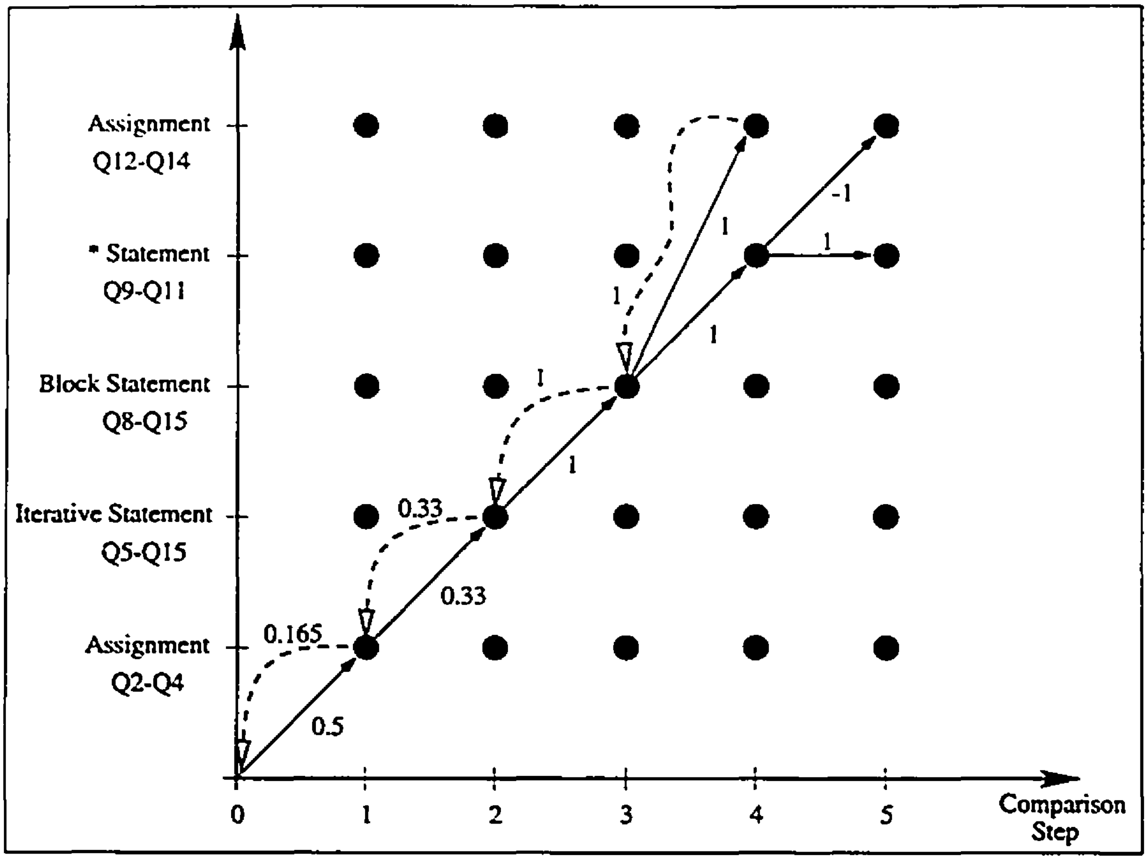


Figure C.2: Comparison steps in the Viterbi algorithm for the example.

Finally the calculated probability is assigned to the final transition to the *Last Sentinel* state of the "outermost" APM and a successful code segment localization is reported. The described steps are shown in figure C.2, the dashed line presents the reported path of recognition.

List of Abbreviations

ACL	Abstract Concept Language
AST	Abstract Syntax Tree
APM	Abstract Pattern Model
BNF	Backus Normal Form
CLIPS	C Language Integrated Production System
CSL	Code Segment Localizer
GUI	Graphical User Interface
HMM	Hidden Markov Model
HTML	Hyper Text Markup Language
KLOC	Kilo Lines Of Code
LOC	Lines Of Code
MLOC	Million Lines Of Code
NRC	National Research Council of Canada
REVENGE	REVerse ENGineering Environment
SCM	StatiC Model
SQL/DS	Structured Query Language/Data System
TMB	Telos Message Bus

Bibliography

[1] J.B. Arseneau. *Software Reengineering & Maintenance Tools.* http://www.erg.abdn.ac.uk/users/brant/src/tools.html.

[2] K. Bennett. "Legacy systems : Coping with success". *IEEE Software*, January 1995.

[3] T.J. Biggerstaff. "Design recovery for maintenance and reuse". *IEEE Computer*, pages 36–44, July 1989.

[4] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. "Program understanding and the concept assignment problem". *Communications of the ACM*, 37(5):72–82, May 1994.

[5] D.C. Bliar and S.D. Lee. "An evaluation of retieval effectiveness for a full-textdocument retrieval system". *Communications of the ACM*, 28(3):289–299, 1985.

[6] G. Booch. *Object-Oriented Analysis and design.* The Benjamin/Cummings Publishing Company ,Inc, 1994.

[7] B. Britcher and J. Craig. "Upgrading aging software using modern software engineering practices". In *IEEE Conference on Software Maintenance*, pages 162–170, 1985.

[8] Intersolv Sales Brochure. *Design Recovery for Excelerator.* 1991.

[9] D.C. Brotsky. *An Algorithm for Parsing Flow Graphs.* Master's thesis, MIT, 1984.

[10] E. Bush. "The automatic restructuring of cobol". In *IEEE Conference on Software Maintenance*, pages 35–41, 1985.

[11] E. Buss and J. Henshaw. "A software reverse engineering experience". In *CASCON*, pages 55–73. IBM Canada Ltd, October 1991.

[12] CASE. "Reengineering and maintenance". In *CASE Outlook*, 1989.

[13] E.J. Chikofsky and J.H. Cross II. "Reverse enginnering and design recovery a taxonomy". *IEEE Software*, pages 13-17, January 1990.

[14] N. Cooke and W. Schvaneveldt.. "Effects of computer programming experience on network representation of abstract programming concepts". *International Journal of Man Machine Studies*, 29:407-427, 1988.

[15] T.A. Corbi. "Program understanding:challenge for the 1990's". *IBM Systems Journal*, 28(2):294-306, 1989.

[16] J.R. Cordy and I.H. Carmichael. *The TXL Programming Language, Syntax and Informal Semantics - Version 7.* Technical Report Technical Report 93-355, Dept. of Computing and Information Science, Queen's University, 1993.

[17] Legasys Corporation. *TXL Transformation System.* http://www.qucis.queensu.ca/home/cordy/legasys.html.

[18] B.K. Das. "A knowledge based approach to the analysis of code and program design language". In *IEEE Conference on Software Maintenance*, pages 290-296, 1989.

[19] S. Davies. "The nature and development of programming plans". *International Journal on Man Machine Studies*, 32:461-481, 1990.

[20] R. Dekker and F. Ververs. *A design recovery prototype.* Technical report, Delft University of Technology, 1995.

[21] E. Buss et al. "Investigating reverse engineering technologies for the cas program understanding project.". *IBM Systems Journal*, 33(3):477-499, 1994.

[22] F.W. Callics et al. "A knoweldge based system for software maintenance". In *IEEE Conference on Software Maintenance*, pages 319-324, 1988.

[23] G. Arango et al. "Maintenace and porting of software by design recovery". In *IEEE Conference on Software Maintenance*, pages 42-49, 1985.

[24] J. Mylopoulos et al. "Telos: Representing knowledge about information systems". *ACM Transactions on Information Systems*, pages 325-362, October 1990.

[25] K. Kontogiannis et al. "The development of a partial design recovery system for legacy systems". In *CASCON*, pages 206-216, October 1993.

[26] K. Kontogiannis et al. "Pattern matching for clone and concept detection".. In *Journal of Automated Software Engineering*, pages 275-307, 1995.

[27] L.D. Landis et al. "Documentation in a software maintenance environment". In *IEEE Conference on Software Maintenance*, pages 66–73, 1988.

[28] P. Benedusi et al. "A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance". In *IEEE Conference on Software Maintenance*, pages 180–189, 1989.

[29] P. Brown et al. "Class-based n-gram models of natural language". *Journal of Computational Linguistics*, 18(4):467–479, December 1992.

[30] M.T. Harandi and J.Q. Ning. "Knowledge-based program analysis". *IEEE Software*, pages 74–81, January 1990.

[31] J. Hartman. *Automatic Control Understanding for Natural Programs*. PhD thesis, University of Texas at Austin, May 1991.

[32] J. Hartman. "Understanding natural programs using proper decomposition". *Proceedings of the 13th International Conference of Software Engineering*, May 1991.

[33] Imagix. *Imagix 4D*. http://www.teleport.com/ imagix/.

[34] J.H. Johnson. "Identifying redundancy in source code using fingerprints". In *CASCON*, pages 171–183. IBM Canada Ltd., November 1992.

[35] W.L. Johnson and E. Soloway. "Proust: Knowledge-based program understanding". *IEEE Transactions on Software Engineering*, pages 267–275, March 1985.

[36] V. Karakostas. "The use of application domain knowledge for effective software maintenance". In *IEEE Conference on Software Maintenance*, pages 170–176, 1990.

[37] M.A. Ketabchi. "Object oriented intergrated software abalysis and maintenance". In *IEEE Conference on Software Maintenance*, pages 60–62, 1990.

[38] K. Kontogiannis. "Toward program representation and program understanding using process algebras". In *CASCON*, pages 299–317, November 1992.

[39] G.B. Kotik and L.Z. Markosian. *Automating Software Analysis and Testing Using a Program Transformation System*. Technical report, Reasoning Systems Inc., 1989.

[40] S. Letovsky. *Plan Analysis of Programs*. PhD thesis, Yale University Dept. of Computer Science, December 1988.

[41] K.J. Lieberherr and I.M. Holland. "Tools for preventing software maintenance". In *IEEE Conference on Software Maintenance*, pages 2-13, 1989.

[42] R.C. Linger. "Software maintenance as engineering discipline". In *IEEE Conference on Software Maintenance*, pages 292-297, 1988.

[43] Z.Y. Liu, M. Ballantyne, and L. Seward. *An Assistant for Re-Engineering Legacy Systems*. http://www.spo.eds.com/edsr/papers/asstreeng.html.

[44] Lockheed. *InVision*. http://www.lmsc.lockheed.com/newsbureau/pressreleases/9522.html, 1996.

[45] J. Meekel and M. Viala. "Logiscope : A tool for maintenance". In *IEEE Conference on Software Maintenance*, pages 328-334, 1988.

[46] H.A. Muller. *Rigi - A Model for Software System Construction, Intergration and Evolution Based on Module Interface Specifications*. PhD thesis, Rice University, August 1986.

[47] W.M. Osborne and E.J. Chikofsky. "Fitting pieces to the maintenance puzzle". *IEEE Software*, pages 11-12, January 1990.

[48] D. Ourston. "Program recognition". *IEEE Expert*, 4(4):36-49, Winter 1989.

[49] M.C. Overstreet, J. Chen, and F. Byrum. "Program maintenance by safe transformations". In *IEEE Conference on Software Maintenance*, pages 118-123, 1988.

[50] G. Parikh and N. Zvegintzov. *The World of Software Maintenance*, chapter 1, pages 1-3. CSPress, Los Alamitos, 1983.

[51] S. Paul and A. Prakash. "Source code retrieval using programming patterns". In *IEEE Transactions on Software Engineering*, pages 227-242, 1994.

[52] J. Picone. "Continuous speech recognition using hidden markov models". *IEEE ASSP MAGAZINE*, pages 26-41, July 1990.

[53] W. Pree, D. Gangopadhyay, and A. Schappert. "Report on the workshop framework-centered software development". In *Addendum to the Proceedings OOPSLA '95*, pages 100-103, October 1995.

[54] R. Prieto-Diaz. "Domain anlysis an introduction". *Software Engineering Notes*, 15(2):47-54, April 1990.

[55] A. Quilici. "Reverse engineering of legacy systems: A path toward success". In *International Conference on Software Engineering*, pages 333-336, 1995.

[56] A. Quilici and J. Khan. "Extracting objects and operations from c programs". In *Workshop Notes, AI and Automated Program Understanding, AAAI'92*, pages 93 97, 1992.

[57] S.P. Reiss. "Pecan: Program development systems that support multiple views". In *ICSE-7*, pages 324-333, 1984.

[58] C. Rich and L.M. Wills. "Recognizing a program's design : A graph-parsing approach". *IEEE Software*, pages 82-89, January 1990.

[59] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[60] D.B. Smith and P.W. Oman. "Software tools in context". *IEEE Software*, pages 15-19, May 1990.

[61] C. Smythe, A. Colbrook, and A. Darlison. "Data abstraction in a software reengineering reference model". In *IEEE Conference on Software Maintenance*, pages 2-11, 1990.

[62] H.M. Sneed. "Planning the reengineering of legacy systems". *IEEE Software*, pages 24-34, January 1995.

[63] H.M. Sneed and G. Jandrasics. "Software recycling". In *IEEE Conference on Software Maintenance*, pages 82-90, 1987.

[64] AG Software. *FULCRUM 2000.* http://www.saguk.co.uk/web/year2000.html.

[65] E. Soloway and K. Ehrlich. "Empirical studies of programming knowledge". *IEEE Transactions on Software Engineering*, pages 595-609, 1984.

[66] Logic Technologies. *LogiCASE.* http://www.provantage.com/DE_08063.HTM.

[67] Leverage Technologists. *Tools.* http://stout.levtech.com/home.html.

[68] A.J. Viterbi. "Error bounds for convolutional code and asymptotic optimum decoding algorithm". *IEEE Transactions on Information Theory*, 13(2):336-342, 1967.

[69] W.B. Weide, D.W. Heym, and E.J. Hollingsworh E.J. "Reverse engineering of legacy code exposed". In *International Conference on Software Engineering*, pages 327-331, 1995.

[70] L.M. Wills. *Automated Program Recognition.* Master's thesis, MIT, 1987.

[71] L.M Wills. "Automated program recognition: Breaking out of the toy program rut". In *Workshop Notes, AI and Automated Program Understanding, AAAI'92*, pages 129–133, 1992.