INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

PATTERN MATCHING TECHNIQUES FOR PROGRAM UNDERSTANDING

Konstantinos A. Kontoyiannis

School of Computer Science
McGill University, Montréal

November 1996

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

To my parents, Thanos and Ioanna,

who always stood by me.

ABSTRACT

When a successful software system is maintained and evolved for an extended period of time, original design documents become obsolete and design rationales become lost, so reverse engineering activities to reconstruct such information become critical for the software's continued viability.

Pattern matching provides a solid framework for identifying higher level abstractions that may be instances of predefined plans (commonly used algorithms and cliches), programming concepts, or abstract data types and operations. This thesis discusses two types of pattern-matching techniques developed for plan recognition in Program Understanding.

The first type is based on Software Metrics and Dynamic Programming techniques that allow for statement-level comparison of feature vectors that characterize source code program statements. This type of pattern matching is used to identify similar code fragments, and code cloning, facilitating thus code modularization, code restructuring and efficient localization of the occurrence of similar programming errors.

The second type addresses the problem of establishing correspondences, between a parse tree of a custom abstract description language developed (ACL) and the parse tree of the code. Matching of abstract representations and source code representations involves alignment that is again performed using a Dynamic Programming algorithm that compares feature vectors of abstract descriptions, and source code. The use of a statistical formalism allows a score (a probability) to be assigned to every match that is attempted. Incomplete or imperfect matching is also possible leaving to the software engineer the final decision on the similar candidates proposed by the matcher.

The system has been implemented to analyze software systems written in PL/AS and C.

RÉSUMÉ

Lorsqu'un système de logiciel évolue et est mis à jour sur une longue période de temps, les documents originaux sur sa conception deviennent périmés et les raisons de sa conception peuvent être perdues: il devient alors critique de procéder à des activités de génie inverse (reverse engineering) pour assurer la viabilité continue du logiciel.

Le filtrage (pattern matching) fournit un cadre solide pour l'identification d'abstractions de haut niveau qui pourraient être des instances de plans prédéfinis (algorithmes courramment uitlisés et clichés), de concepts de programmation, ou de types de données abstraits et leurs opérations. Cette thèse traite de deux types de techniques de filtrage dévelloppées pour la reconnaissance de plans en Compréhension des Programmes.

Le premier type est basé sur des techniques de Métriques de Logiciels (Software Metrics) et de Programmation Dynamique qui permettent la comparaison au niveau des énoncés de vecteurs de traits qui caractérisent les énoncés de code source des programmes. Ce type de filtrage est utilisé pour identifier les fragments de code similaires, et le clonage de code, facilitant ainsi la modularisation du code, la restructuration du code et la localisation efficace d'instances d'erreurs de programmation similaires.

Le second type traite du problème de la mise en place de correspondences entre un arbre d'analyse syntactique d'un langage abstrait de description dévelopé (ACL) et l'arbre d'analyse syntactique du code. Le filtrage de représentations abstraites et de représentations de code source requiert leur allignement, ce qui est de nouveau fait avec un algorithme de Programmation Dynamique qui compare les vecteurs de traits des descriptions abstraites et du code source. L'emploi d'un formalisme statistique permet d'affecter un pointage (une probabilité) à chaque filtrage tenté. Un filtrage incomplet ou imparfait est aussi possible, laissant à l'ingénieur en logiciel la décision finale sur les candidats similaires proposés par le filtreur.

Le système a été implémenté pour analyser des systémes de logiciel écrits en PL/AS et C.

ACKNOWLEDGEMENTS

During my studies at McGill I had the opportunity to meet different people from whom I learned many lessons in science and in human relations. It was a wonderful journey for me. For this journey, I would like to thank my supervisors Renato DeMori, and Ettore Merlo for giving me the opportunity to explore the frontiers of science and be part of an exciting team. I thank them also for providing always support and encouragement throughout my studies. I would like also to thank the Principal Investigators of the team I worked with, namely John Mylopoulos, Hausi Müller, Jacob Slonim, and the members of IBM Canada, Centre for Advanced Studies, for providing a very fertile research environment.

But it was not only the exciting research environment that made my studies a memorable experience. Many friends, and fellow students helped me understand and appreciate the simple things in life that happen every day and pass unnoticed to most of us. Among those I thank Yiannis Rekleitis, Christos Magdalinos, and the members of the Speech Lab for being so supportive.

I thank my friends Nikos Krivossidis, Bassem Khalife, Mariza Komioti and Stergios Anastasiadis, with whom I shared the first years as a student in Montreal. Finally, I would like to thank Jasmina for giving me the energy and the support to finish this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PUBLICATIONS

List of Publications Relevant to this Thesis

- Refereed Journals

(i) "Pattern Matching for Clone and Concept Detection" K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, M.Bernstein, *Journal of Automated Software Engineering*, vol.3, pp. 77-108, 1996 This publication describes the core of this Ph.D thesis. My contribution in this paper was the design and development of a scalable method for localizing similar programming patterns and for measuring of sctructural similarity between code fragments. It uses Dynamic Programming and Markov Models to establish a structural similarity probability. I shared interesting discussions with the rest of the authors who had experience in the field of applying Markov models in Speech Recognition.

(ii) "Reengineering User Interfaces", Merlo, E., Gagnie, P.Y., Girard, J.F., Kontogiannis, K., Hendren, L., Panangaden, P., DeMori, R. *IEEE Software, January 1995* This publication discusses the reengineering of CICS line-based user interfaces and the migration process towards modern designs (i.e X-Windows). This research has been quoted as a standard in Michael Brodie's book "Migrating Legacy Systems", by Morgan Kaufman Publishers, 1995, and in ACM Communications special issue on Reverse Engineering, vol. 37, No. 5, 1994. pp. 84-93. My participation on the project focused on the design of an abstract user interface specification language using Process Algebras.

(iii) "Investigating Reverse Engineering Technologies for the CAS program Understanding Project", Buss, E., DeMori, R., Gentleman, M., Henshaw, J., Johnson, H., Kontogiannis, K., Merlo, E., Muller, H., Mylopoulos, J., Paul, S., Prakash, A., Stanley, M., Tilley, S., Troster, J., Wong, K. *IBM Systems Journal, Vol. 33, No. 3, 1994*
This publication discusses the techniques and the results from our CRD project with IBM Canada. My contribution on this project was the development of techniques to

identify error prone code and redundant code in large software systems using software metrics. The target legacy system was SQL/DS which is written in a PL-type language.

- **Refereed Conferences**

  (i) "A Generic Integration Architecture for Cooperative Information Systems", John Mylopoulos, Avi Gall, Kostas Kontogiannis, Martin Stanley, In Proceedings of COOPIS '96, Brussels, Belgium

  This paper discusses issues of software tool integration. It examines the Event-Condition-Action principle of active data bases and investigates its use for the design of a data and control integration environment. My contribution on this research was the design of the requirements for a dynamic CASE integrated toolset. This reseacrh capitalized on the expertise gained during our IBM/CRD project and the expertise of Avi Gall and John Mylopoulos on active data bases.

  (ii) "Pattern Matching for Design Concept Localization", Kontogiannis, K., DeMori, R., Merlo, E., Bernstein, M., Galler, M. *Working Conference on Reverse Engineering WCRE'95, July 1995, Toronto, ON.*

  This publication was ranked among the best in WCRE'95 and was invited to be expanded in a Journal publication in the journal of Automated Software Engineering. It discusses the application of a stochastic pattern matcher using Markov Models for code segmentation and plan localization. The significance of this approach is that it is scalable and efficient on the time and space resources it requires.

  (iii) "Localization of Design Concepts in Legacy Systems", Kontogiannis, K., DeMori R., Bernstein, M. *Proceedings of the International Conference on Software Maintenance 1994, Victoria, BC.* This publication marked the first findings from the application of Dynamic Programming in a pattern matching tool targeting clone recognition. The paper provides insights for a scalable pattern matching approach that led to the development of a prototype system for plan recognition.

  (iv) "The Development of a Partial Design Recovery Environment for Legacy Systems", Kontogiannis, K., Bernstein, M., Merlo, E., DeMori, R. *Proceedings of CASCON'93, Toronto, ON.*

  (v) "Program Representation and Behavioural Matching For Localizing Similar Code Fragments", Kontogiannis, K. *Proceedings of CASCON'93, Toronto, ON.*

(vi) "Reverse Engineering of User Interfaces", Merlo, E., Girard, J.F., Kontogiannis, K., Panangaden, P., DeMori, R. *Proceedings of Working Conference on Reverse Engineering, WCRE'93, Maryland, Baltimore*

This paper discussed initial findings on the DMR project. Please refer to the relevant *IEEE Software* paper above.

- **Refereed Workshops**

(i) "Partial Matching for Code Similarity", *International Conference on Artificial Intelligence, Workshop on Software Engineering and A.I, August 1995, Montreal, QUE.* DeMori, R., Kontogiannis, K.

This paper discussed the use of A.I modeling and matching techniques in Program Understanding. The motivation for this paper was the potential use of A.I techniques to handle complexity issues related to code segmentation and plan recognition. The paper investigates the use of pattern matching techniques and modeling techniques in the area of Software Engineering.

(ii) "User-Assisted Design Recovery of Legacy Software Systems", Kontogiannis, K., Tilley, S., DeMori, R., Muller, H. *Workshop on the Intersection of Software Engineering and Artificial Intelligence, ICSE'16, Sorrento, Italy, May 1994*

Similar as above.

(iii) "A Process Algebra Based Program and System Representation for Reverse Engineering", Merlo, E., DeMori, R., Kontogiannis, K. *Proceedings of the Workshop of Program Comprehension, Capri 1993, Italy*

This paper discussed the usefulness of formal methods and in particular Process Algebras to represent the semantics and the behaviour of programs. The paper investigated the use of bisimulation for assuming behavioural similarity between two code fragments. A prototype system has been demonstrated that allowed for plan recognition of small size systems ($\prec$ 1KLOC). This paper describes my initial approach to the problem of Program Understanding within the scope of my Ph.D research. This approach has been demonstrated to be very accurate, allowing for matching under syntactic or implementation variations. The drawback was that it did not scale up properly and required significant effort to model source code as Process Algebra equations.

CHAPTER 1

Reverse Engineering

1. Introduction

Despite the fact that faster and cheaper computer hardware continues to appear on the market at an impressive rate, much of the software currently used is on average ten to fifteen years old [**Osborne90**]. In most cases these programs have to be corrected, migrated to new platforms, or enhanced in performance. Usually, people involved in these processes are not the original system designers, and they have to devote a significant amount of resources in order to understand the system to be maintained. An important point is that a maintainer not only has to understand the code, but also the system as a whole, its functions, its environment, its subprogram structure, its data base set up, and many other factors [**Wedo85**].

The objective of reverse engineering is the development of a set of tools and techniques for understanding unfamiliar code, so that system maintenance can be facilitated [**Chiko90**]. Having obtained enough information on the system a maintainer can proceed with the restructuring phase where the system is enhanced or adapted to a new environment. The amount of software maintenance research in the last three years has increased [**Hale90**], and it is estimated that in the U.S. alone 2% of the country's gross national product was spent in 1985 for software maintenance [**Karakostas90**]. In the past few years several research groups have focused their efforts on the development of tools for program understanding and program restructuring. The major research issues involve the development of formalisms to represent program structure, control and data flow, as well as visualizing program execution.

In [**Chiko90**] a taxonomy of reverse engineering terms is given. This taxonomy, divides Reverse Engineering in two major subareas : *a)* Re-documentation and; *b)* Design recovery. Re-documentation is a process in which alternative views of the program are presented in order to

reflect certain characteristics of the subject system. Design recovery is a subset of reverse engineering in which abstractions of the subject system are created in order to impose a meaning on a program segment. Restructuring and re-engineering are two other terms which are relatively close in meaning. Restructuring is the transformation of a software system without affecting its functionality, while re-engineering is the transformation of a subject system while adding new functionalities to the system.

Reverse engineering is the process of analyzing a subject system to:

- identify the system's component's and their interrelationships, and
- create representations of the system in another form or at a higher level of abstraction.

Reverse engineering does not involve changing the subject system; it is process of examination. Our work will focus on the area of program understanding and in particular the area of design recovery.

The term design recovery means understanding the program as a whole with respect to functional specifications, input parameters, expected output, performance, as well as on the software and hardware environment in which it runs. The available information is usually formal (source code) and informal (comments). Source code is represented usually as an ASCII file. This file may contain a number of hints about the run-time functionality of the program (how and in what order modules are invoked), parameter passing, aliases, side effects etc. On the other hand, informal information is a valuable source of information for the task of understanding complex features involving the understanding of the organization of program structure (how procedures or submodules are organized). Hints may be in the form of comments, *I/O* messages, meaningful variable names, documentation etc.

2. Design Recovery

According to Biggerstaff [**Biggerstaff89**]:

> Design recovery recreates design abstractions from a combination of code, existing design documentation, personal experience, and general knowledge about problem and application domains. Design Recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code.

According to Harandi and Ning [**Harandi90**] to maintain a program, a programmer needs to develop a mental model of its function first. To facilitate this, four different program views are provided.

(i) The implementation-level view is represented as an abstract syntax tree and a symbol table of program tokens.

(ii) The structure-level view gives an explicit representation of the dependencies among program components.

(iii) The function-level view relates parts of the program to their functions and shows the logical relations among them.

(iv) The domain-level replaces items in the function-view by concepts specific to the application domain.

Understanding the code includes code representation, structural representation of the system (modules' interaction), data flow, control flow graphs, conceptual abstraction of the code (specifications, abstract data types, normalization). However, understanding the code with an *automated* system, usually requires access to the domain model, and the domain information [**Prieto-Diaz90**]. However, complete automated design recovery is not always feasible. In this case partial design recovery is a more realistic objective. Partial solutions can be very useful, as the maintainer can fill in the uninterpreted gaps using his own programming skills and experience.

3. Thesis Objectives

3.1. Motivation and Background. Large-scale production software systems are expensive to build and, over their useful lifetimes, are even more expensive to maintain. Successful large-scale systems are often called "legacy systems" because (a) they tend to have been in service for many years, (b) the original developers, in the normal course of events move on to other projects, leaving the system to be maintained by successive generations of maintenance programmers, and (c) the systems themselves represent enormous, irreplaceable corporate assets.

Legacy systems are intrinsically difficult to maintain because of their sheer bulk and because of the loss of historical information: design documentation is seldom maintained as the system evolves. In many cases, the source code becomes the sole repository for evolving corporate business rules.

During system maintenance, it is often necessary to move from low, implementation-oriented levels of abstraction back to the design and even the requirements levels. The process is generally known as "reverse engineering"[1].

In particular, it has been estimated that 50 to 90 percent of the maintenance programmer's effort is devoted to simply understanding relationships within the program. The average Fortune 100 company maintains 35 million lines of source code (MLOC) with a growth rate of 10 percent

[1] In this thesis, "reverse engineering" and related terms refer to legitimate maintenance activities based on source-language programs. The terms do not refer to illegal or not ethical activities such as the reverse compilation of object code to produce a competing product.

per year just in enhancements, updates, and normal maintenance. Facilitating the design recovery process can yield significant economic savings.

We believe that maintaining a large legacy software system is an inherently human activity that requires knowledge, experience, taste, judgement and creativity. For the foreseeable future, no single tool or technique will replace the maintenance programmer nor even satisfy all of the programmer's needs. Evolving real-world systems requires pragmatism and flexibility.

It has been argued [**Wills92**] that programmers use patterns and expert knowledge to recognize programs and programming structures. These patterns, idioms, and commonly used structures are called *plans*[**Huff89**] [**Wills93**]. A `plan` is a commonly used idiom or algorithm in a software system that implements a particular task, a generic or domain concept, or a business rule. A plan can be represented in different levels of abstraction. At the lower level a plan instance is described in terms of its source code implementation. At an intermediate level it is represented as pseudo-code with links and references to actual source code and informal information. At a higher level of abstraction the plan is codified in a Knowledge Representation (KR) formalism that captures the knowledge the experienced programmers use to recognize such plans.

When a group of developers are given the task to understand and maintain a large software system the following activities have to be performed:

- Represent the source code at a higher level of abstraction
- Identify the basic physical structure of the system
- Decompose the system into modules based on data and control flow properties
- Localize particular plans in the code and attach concepts to them
- Identify parts of the system that interact, depend or alter a recognised concept

Recent studies in Reverse Engineering have proposed a number of research issues. Most of them are related to program and knowledge representation methods, search techniques, compiler technology as well as program specification and verification methods.

Design Recovery has been viewed mostly as a program representation and a plan localization problem where representations of the source code are matched against programming plans stored in a static library. In real applications though, the limited number of programming plans that can be encoded, knotty program representation schemes and complex plan localization algorithms make design recovery applications rigid and difficult to be extended to large systems. Thus, it is more realistic to take an approach in which methods and tools for partial design recovery are conceived based on a set of strategic but specific objectives which are dictated by the user and the type of analysis he or she performs.

That introduces the idea of *Goal Directed Design Recovery* in which the appropriate representation method, the level of abstraction, the appropriate analysis tool, and the control strategy are dictated by the objectives and the specific program design attributes the maintainer sets or wants to recover respectively.

In this thesis we focus on the development of pattern-matching techniques that allow for `plan` localization and recognition. Our work addresses the problem of plan localization in two levels.

The first level addresses code to code matching. At this level we devised techniques to compute dissimilarity distances between two code fragments, allowing for the detection of potential code cloning and, the localization of similar patterns in the code that may implement a particular plan.

The second level addresses the problem of localizing abstract code descriptions in a software system.

In particular, this thesis discusses

(i) The development of techniques for source code to source code matching for detecting code duplication and devise dissimilarity distances between two code fragments

(ii) The development of an abstract language to represent programming plans and the corresponding techniques to localize these abstract descriptions in a large software system

(iii) The development of a software framework that allows for CASE tool integration in a distributed environment.

3.2. Thesis Contributions.

3.2.1. *The Code Cloning Problem.* Source code cloning occurs when a developer reuses existing code in a new context by making a copy that is altered to provide new functionality. The practice is widespread among developers and occurs for several reasons: making a modified copy may be simpler than trying to exploit commonality by writing a more general, parameterized function; scheduling pressures may not allow the time required to generalize the code; and efficiency constraints may not admit the extra overhead (real or perceived) of a generalized routine.

In the long run, code cloning can be a costly practice. Firstly, it results in a program that is larger than necessary, increasing the complexity that must be managed by the maintenance programmer and increasing the size of the executable program, requiring larger computers. Secondly, when a modification is required (for example, due to bug fixes, enhancements, or changes in business rules), the change must be propagated to all instances of the clone. Thirdly, often-cloned functionality is a prime candidate for repackaging and generalization for a repository of reusable components which can yield tremendous leverage during development of new applications.

The thesis introduces new techniques for detecting instances of source code cloning. Program features based on software metrics are proposed. These features apply to basic program segments

like individual statements, `begin-end` blocks and functions. Distances between program segments can be computed based on feature differences. We propose two methods for addressing the code cloning detection problem.

The first is based on direct comparison of metric values that classify a given code fragment. The granularity for selecting and comparing code fragments is at the level of a source code statement. This method returns clusters of statements that may be products of cut-and-paste operations.

The second is based on a new Dynamic Programming (DP) technique that is used to calculate the best alignment between two code fragments in terms of *deletions*, *insertions* and, *substitutions*. The granularity for selecting code fragments for comparison is again at the source code statement level. Once two statements have been selected they are compared using their corresponding feature vectors. This method returns clusters of statements that may be products of cut-and-paste operations. The DP approach provides in general, more accurate results (i.e. less false positives) when comparing two blocks of source code statements, than the one based on direct comparison of their metric values. The reason is that using DP comparison occurs at the statement level and informal information is also taken into account (i.e. variable names, literal strings and numbers).

3.2.2. *The Concept Recognition Problem.* Programming concepts are described by a concept language. A concept to be recognized is a phrase of the concept language. Concept descriptions and source code are parsed. The concept recognition problem becomes the problem of establishing correspondences, as in machine translation, between a parse tree of the concept description language and the parse tree of the code.

A new formalism is proposed to see the problem as a stochastic syntax-directed translation. Translation rules are pairs of rewriting rules and have associated a probability that can be set initially to uniform values for all the possible alternatives.

Matching of concept representations and source code representations involves alignment that is again performed using a dynamic programming algorithm that compares feature vectors of concept descriptions, and source code.

The proposed concept description language, models *insertions* as wild characters (*Abstract Statement** and *Abstract Statement*$^+$) and does not allow any *deletions* from the pattern. The comparison and selection granularity is at the statement level. Comparison of a concept description language statement with a source code statement is achieved by comparing feature vectors (i.e. metrics, variables used, variables defined and keywords).

Given a concept description $\mathcal{M} = A_1; A_2; ..A_m$, a code fragment $\mathcal{P} = S_1; S_2; ..S_k$ is selected for comparison if: *a)* the first concept description statement A_1 matches with S_1, and *b)* the sequence of statements $S_2; ...S_k$, belong to the inner most `begin-end` block containing S_1.

The use of a statistical formalism allows a score (a probability) to be assigned to every match that is attempted. Incomplete or imperfect matching is also possible leaving to the software engineer the final decision on the similar candidates proposed by the matcher. A way of dynamically updating matching probabilities as new data are observed is also suggested by the use of a cache.

3.2.3. *Tool Integration.* A suite of complementary tools from which the programmer can select the most appropriate one for the specific task at hand. An integration framework enables exploitation of synergy by allowing communication among the tools.

This work has been incorporated on a (*R*everse *E*ngineering *E*nvironment), based on an open architecture for integrating heterogeneous tools. The tool-set is integrated through a common repository specifically designed to support design recovery [**Mylo96**] [**Buss94**]. Individual tools in the kit include Ariadne [**Konto94**] [**Konto96a**], ART [**Johnson94a**] [**Johnson94b**], and Rigi [**Tilley95**] [**Muller93**]. ART (*A*nalysis of *R*edundancy in *T*ext) is a prototype textual redundancy analysis system. Ariadne is a set of pattern matching and design recovery programs implemented using a commercial tool called The Software Refinery [2]. Rigi is a programmable environment for program visualization. The tools communicate through a flexible object server and single global schema implemented using the Telos information modelling language and repository [**Mylo96**].

[2] "The Software Refinery" and REFINE are trademarks of Reasoning Systems, Inc.

CHAPTER 2

State-of-the-Art and Practice on Design Recovery

1. Representation Methods

In order to recover the design of an unknown program, source code has to be represented in a higher level of abstraction. Such source code representation must reflect more the design activity than the source code itself. Most of the research approaches addressing the source code representation problem, focus on the development of mathematical formalisms, and techniques which can facilitate:

(i) representation of program functions in a more abstract way than source code [**Letovsky88**],

(ii) the representation of the behavior of a program [**Hoare85**], [**Milner89**], [**Stoy77**], [**Scott76**], [**Hennessy91**]

(iii) search techniques (borrowed from A.I, or graph theory), [**Rich90**], [**Engberts91**], [**Ning94**], [**Johnson85**],

(iv) ways to reflect information on the problem and the application domain [**Biggerstaff94**],

(v) user friendliness, in terms of how program design is presented to the programmer [**Muller91**],

(vi) adaptability and modeling, in terms of how easily one representation can be transformed into another more abstract one (in case of reverse engineering) or less abstract (in case of forward engineering), and

(vii) portability to a computer environment (be able to define data structures for encoding the formalism).

1.1. Methods of Internal Representation of the Source Code . In the literature one can find a variety of methods for representing source code for software maintenance purposes. It is widely accepted [**Sneed88**] that models for viewing software consists of three levels:

(i) the specification level,

(ii) the design level, and

(iii) the coding level.

The focus of Design Recovery is to provide means to achieve a representation of the program at the Design Level.

In some applications [**Sneed88**], the program is "horizontally" partitioned into divisions. Each division contains information for different parts of the program. For example, in the case of a COBOL program, the Identification division may contain the program name and some general information, the Environment division may contain information of how the system is connected to the physical environment or the platform it runs and so on. Other divisions are more relevant for Design Recovery, namely the Data division and the Procedure division. The Data Division contains the description of the data structures (both external and internal), and the Procedure division contains the executable statements grouped into Sections or Paragraphs. Data structures can be represented in a variety of ways. One way is the tabularization [**Sneed87**] where a table is defined with an entry for each data structure encountered, recording its name, position, type, length, dimension, usage etc. Using such a tabular representation, some groups [**Overstreet88**] use transitive closure algorithms to compute data-flow and variable dependencies. For the procedure level the same technique can be used by defining Relationship Matrixes, for representing relationships among procedures, variable referencing or, relationships between constants, variables, and procedures within the same module (scope of variables).

Another approach for representing source code is using Decomposition Hierarchies [**Ligner88**], [**Hartman91b**]. This technique is based on the Structure Theorem [**Ligner88**], which states that any proper program (single entry, single-exit programs) can be represented as a structure consisting only of primitive programs (sequence, *if/then* structures and loop structures), which some authors [**Bush85**] call normal forms. This approach has the advantage that one can define an *equivalence* mapping for transforming the original unstructured code into a structured one. The first step in this approach is to create an Abstract Syntax Tree of the original source code. Subsequently, tree to tree transformations can be applied in order to obtain a tree from which reduced control flow expressions can be obtained such that the resulting tree can be transformed into a directed graph containing only normal forms. Programming clichés [**Rich90**] can be used for recognizing pieces of code. The transition from the internal representation of code into higher level conceptual program abstractions can also be obtained by extracting conceptual representations by deduction, applying a graph grammar or a pattern matching algorithm. Dependency analysis tools can be used to enhance the internal representation of the code. Dependency analysis [**Wilde89**] is based on graphs which represent definition dependencies (when a program entity is used to define another), calling dependencies (when one module calls another), functional dependencies (when a data object

is created or updated) and finally data flow dependencies [**Gallagher91**] (when the value of one object is used to calculate the value of another).

A third class of methods for representing code uses a unified object-based representation of code segments [**Das89**], [**Murray88**]. In this approach, basic language constructs (e.g. while loops) are represented as objects and their syntax is captured as a list of attributes (e.g. logical condition, body, etc.). The actual source code is represented by instances of these generic objects, and attribute values may refer to other instances (e.g. a body of a loop may be another loop). High level programming knowledge can be encapsulated in rules. A set of such rules can be used to reason about the program (e.g. finding syntactic or non-syntactic bugs). Thus, programs may be viewed as object bases. In some systems [**Ketabchi90**], these objects can be created partially automatically, from the Backus Normal Form (BNF) description of the language. Instances of generic objects can be created by a scanner, a parser and a set of semantic action routines. These approaches treat a software system as assemblies, which have multiple aspects such as structure and functionality. A more abstracted view of the above technique is given in [**Holland89**] and [**Landis88**]. Here not only basic language constructs are represented as objects but also more complex constructs, such as functions or program submodules. This method enhanced with basic rules governing dependencies (eg. if *class1* is a client of *class2* then *class1* depends on *class2*) can be used to create dependency graphs, or even allow for automatic recognizing parameterization of programs (write more general programs). This can be achieved since data groupings or abstract data types (queues, stacks etc) can be defined as object-like structures and be used in different applications by creating instances of theirs.

Graphs, as mentioned earlier, provide a natural way to visualize program information. They are adaptable and most important, they are formal mathematical structures. In some approaches [**Colbrook89**] flow graphs are transformed into prime programs (normal form), and then abstracted into more general structures. The process may be described as reducing the program to be understood to small prime programs and then creating, in a step by step process, functions combining them at higher and higher levels until a full specification is achieved. It is obvious that such an approach requires an excessive library of abstract data types. Furthermore, graph complexity can be used as a metric for the maintainability of the code [**McCabe90**]. Similarly, data flow diagrams and structure charts [**Gillis90**] can be generally used to model the data transformation aspects of a software system [**Beneduci89**], since they emphasize the logical flow of data and control while de-emphasizing implementational details and physical solutions of the problem.

In [**Smythe90**], reverse engineering separates into the Encapsulation phase, the Transformation phase, the Normalization phase, the Interpretation phase, the Abstraction phase, and finally the

Causation phase. Basically in these phases the source code is parsed into an intermediate language and the control flow is normalized. At this point, the process of deriving the meaning of a piece of software begins. The code represented in this intermediate language is replaced by logical comments, starting from the inner most blocks and working outward. In the abstraction phase, objects and object hierarchies are identified. Data are mapped onto the procedures so that data operators are separated and grouped with the data they operate upon. Application domains are mapped to objects. In the last phase services which must be provided to the user and constraints which have to be met are identified.

In another approach [**Callics88**] programming knowledge is captured in the form of programming plans [**Rich90**], [**Das89**], [**Davies90**] which are abstract representations of algorithmic structures. A plan lists the building components of an algorithm in terms of atomic program elements or other plans. A plan also identifies the proper sequence of these building components which is defined in [**Harandi90**] and [**Harandi88**] as *event path expression.* Plan definitions are translated by a Plan parser into inference rules as system's understanding knowledge. A pattern directed Inference engine is used for recognizing plans in a program. Moreover, a Justification-based Truth Maintenance System (JTMS) can be used for recording the understanding process. Using this scheme source code has to be paraphrased into a plan and then be matched with the entities in the knowledge base or the predefined plans. The drawback of this approach is that it is not a trivial task to define a system's knowledge as plans, capture all variations of an algorithm, and incorporate appropriate heuristics. Furthermore, there is no guarantee of the completeness and correctness of the knowledge base. Neural nets with learning capabilities could play a role in this approach.

However, it has been found that programmers use many clichés in their programming tasks. This means that they call on their personal library of previously written modules and data structures, for reuse through adaptation for the problem at hand. Abstract and semantic knowledge [**Lebowitz83**] can improve parsing since the syntactic parsing is enhanced with domain knowledge or with programming plans.

Clichés can be recognized in existing programs to recover the programmer's abstract concepts and intentions. According to Hartman [**Hartman91a**], three entities must be present for the uncovering of clichés to be successful:

(i) a program representation or model,

(ii) programming knowledge of standard plans, and

(iii) search and comparison to find a plan instance.

2. Recognition Methods

Once the representation of the basic components of a program by plans, clichés, or other formalisms has been studied, representation and comparison methods for controlling plan detection in complex applications have to be considered.

Comparison methods focus on techniques to perform, for example, simple plan instance recognition, by proving equivalence or other relations, between components. If representation is based on plans, the power of a comparison method is the number of plan instances it can recognize using a given plan and its ability to abstract away from the comparison process implementation details used in different plan instances. Comparison algorithms depend heavily on the program and on the selected representation method and it does not always involve simple pattern matching. Comparison algorithms may apply transformation rules to establish program-plan equivalence. Equivalence based on specific criteria, is the strongest relation one can prove between a code fragment and a programming plan but in practice partial recognition is the best that can be achieved. Partial recognition deals with the problem of recognizing plan instances even when these plans are interleaved with other type of information in the code or they are scattered throughout the program. Multiple, failed or incomplete plan recognition has to be taken into consideration as well. Multiple recognition occurs when a single programming plan matches more than one program part. Ambiguities can be resolved using needs, domain knowledge or information besides the program and plans. On the other hand, failed recognition should produce failure information explaining the cause of failure. The most common case, though, occurs when no success or failure can be proven. In this case, incomplete bindings should be produced for explanation and control.

Basic components are part of a program behavior and are recognized following a certain control strategy. Top level control methods focus on techniques to select program parts and programming plans for comparison in order to achieve plan instance localization. Moreover, top level control applies the comparison results to the application program. Program parts and programming plans, represented at a higher level of abstraction than the source code are selected using a top-level control strategy and used as input to a comparison module. The output of such a comparison are recognized concepts and program parts satisfying the specifications of a programming plan. Control can be guided by the needs of the particular application as well as by the results of previous comparisons. Search algorithms are used to select from the program representation different programming parts for comparison. Bottom-up search strategies systematically select all program parts covering the program representation, while top-down search strategies seek single parts that can be used to satisfy a given expectation (subgoal). Programming plans and program parts are not always represented in the same formalism. Moreover, during the recognition process comparison has to performed between

already recognized concepts and original program material. Hierarchical recognition control strategies focus on such multi-multi-leveled representations and are used for compositional recognition where complex concepts are recognized in terms of their subcomponents.

Furthermore, program decomposition can be used to guide the selection process. Performance is best when decomposition produces program parts which correspond well with the plans in the plan library. Program decomposition can be performed *a priori* before the selection process starts, or in a *dynamic* way based on previous recognition results and the current needs of the application as the selection process is performed.

2.1. Methods Used for Controlling Recognition . In the literature one can find a number of different methods used to guide and perform the plan instance recognition process.

As far as the comparison methods are concerned, some systems (eg. PROUST) [**Johnson85**], [**Engberts91**] match syntax trees with syntax tree templates. A plan matches a program statement if its unified template matches the statement's syntax tree, and its constraints and subgoals are satisfied. TALUS, [**Ourston89**] compares student and reference functions by applying a heuristic similarity measure. In CPU, [**Letovsky88**] programs are represented as lambda calculus expressions and procedural plans. Comparison in CPU is performed by applying a unification and matching algorithm on lambda calculus expressions. In UNPROG, [**Hartman91b**] program control flow graphs and data flow relations are compared with the programming plan's control flow graph and data flow relations. The objective here is proving plan's data flow a subset of a program's part data flow. Quilici, [**Quilici92**] matches frame schema representations of C code. If they structurally match then data flow graphs are compared as well. GRASP, [**Wills92**] uses attributed data flow sub-graphs to represent programs and programming plans. Comparison is performed by matching sub-graphs and by checking constraints involving control dependencies and other program attributes.

As far as the top-level control methods are concerned PROUST, [**Johnson85**] uses a top-down control strategy applied to a solution goal tree. This control strategy, is enhanced with heuristics for ordering, comparison and evaluation. Transformations are applied in order to reduce differences due to implementation variations and bugs. TALUS, [**Ourston89**] uses the A^* best first search algorithm in order to find a mapping between student functions and reference program functions maximizing a heuristic measure. CPU, [**Letovsky88**] 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. Quilici's system applies an indexing scheme to select candidate plans and then performs semantic abstractions by substituting the selected frame with the abstracted one. Hierarchical recognition proceeds upward until no more abstractions can be performed. Finally, GRASP [**Wills92**] 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 of possible interpretations.

3. Informal Information Analysis

The methods presented so far are general purpose and none of them has been proven sufficient to completely solve the design recovery problem for large systems. Other types of analysis can offer interesting solutions, especially when partial design recovery of certain types of applications is the objective. These types of analysis will be briefly reviewed in following paragraphs. They can inspire the conception of useful tools.

One of the most useful sources of information is the documentation and the mnemonic identifier names used throughout the program. Code can be analyzed more easily and parsed so that ASTs can be built and furthermore be abstracted to more general representations so that, the user understands the nature of a design in human terms. Informal information obtained by an analysis of the comments, variable names, and documentation is particularly useful for this purpose. "By restoring the comments from the original code we can elaborate several of our guesses and enhance our understanding of some of the functions and variables" [**Biggerstaff89**]. Informal information provides the means for understanding the *computational intent* of the code in a way that is impossible when plain source code is used. The research issues dealing with this subject are:

(i) how to represent the semantics provided by the informal information,

(ii) how can be related to formal information provided by the code, and

(iii) define a set of operations that integrate this information and implement a design recovery process.

An interesting approach based on automatic learning is described in [**Merlo93**].

4. Interactive Query Capabilities

On designing an interface [**Shneiderman86**], [**Hix89**], for a software maintenance tool there are some points which are important to the end user. One such is the option of interactive query capabilities. The user should be able to interrupt the maintenance system and ask for information, manipulate graphs, choose windows and select different ways to represent data [**Hill87**].

Some of the queries the system should support are:

- asking for information on the data flow graphs,
- asking for information on the control flow graphs,

- program fragment localizations,
- editing and extracting selected parts of code (the ones that are relevant to a selected process),
- redundant and duplicated code detection,
- visualizing aliases,
- dead code detection,
- performing user guided transformations of the graphs,
- selecting domain models (uploading knowledge bases relevant to the domain, etc.).

The previous points can be applied in a more generalized context where the user not only asks for information on the processes the maintenance tool is performing but also is able to access information of the system as a whole. That means he must be able to view data bases, operating system parameters, and module relationships. Moreover, the user should be able to visualize the expected side effects of a change he is going to implement. Call graphs [**Rajlich88**], [**Lieberman84**], dynamic variable bindings, and aliases should be presented. Generally, the tool should support an environment which provides to the user with multipurpose facilities.

5. State of the Practice

The current software maintenance crisis has lead to a large increase in research and development in this area. Research centers are looking for new ways to ease the burden of re-engineering. At the same time, many commercial developers are taking advantage of now highly respected buzzwords *reverse engineering* and *re-engineering*. Most of them provide a sharp user interface based on established principles, generally data and flow control analysis. The utility of these tools should not be underestimated, as this kind of support can greatly increase maintenance efficiency.

Hopefully, the next generation of commercially available re-engineering tools will be based on research that is currently underway, and will provide the magic that is often implied in the sales brochures of today.

5.1. Tools on the Market. Naturally, many software developers have moved to meet the needs of software maintainers. Some of research into reverse engineering has already been translated into commercially available products. REFINE [1] [**Kotik89**] is an interactive software development system which is useful for software analysis and testing. It provides three tools: a high level specification language, an object-oriented database, and a language processing system.

The database is used to store the software as annotated abstract syntax trees, using an object-oriented representation. The specification language is used to query the database. The language processing system must be provided with a description of a language in the form of a grammar

[1] REFINE is a trademark of Reasoning Corp.

together with a language domain model. REFINE also allows programming templates to be defined, which can be used to test whether a program is an instance of a defined template, or to build one which is.

Rules are written which use pattern matching to identify logical code fragments, and bind their text to identifiers. These identifiers may then be used to write out code in a new form. The pattern matching capabilities of REFINE can be used to restructure program statements. Programs are converted between source code and the object-oriented database using the parsers created by the language processing system. The REFINE object system supports a data model that is close to the standard conceptual view of annotated abstract syntax trees. There is a specification language which is used to query and update the database. Once the software has been stored in the database, it may be automatically transformed by defining output rules.

Bachman Information Systems [**Bmc**] provides the following products: BACHMAN / Database Administrator, the DBA Catalog Extract, a Data Analyst, and the DA Capture. These products are capable of reading physical data definitions in a number of different formats and converting them to graphical, logical representations. These representations can be used to modify the database structure, and then the tools can be used to optimize the database design and regenerate it.

Cadre Technologies [**CASE89**] [**Cadr**] has developed a set of applications which can provide statistical information concerning program execution. It also tracks call hierarchy, and can present all its results graphically.

Viasoft [**Viasoft**] has a set of products known as VIA/Center which operate on COBOL programs. The analytical engine can create a database of information about a program's characteristics; like how logic and data are related, and how control is passed between modules. Viasoft also promises more advanced products for the future.

Hypersoft [**HypSoft**] has an application which runs on a VAX called Application Browser. It facilitates navigation through COBOL programs by providing a graphical user interface to the source code.

Ten X Technology [**CASE89**] has a similar product for C programs.

The Intersolv [**Intersolv**] company has produced an application called Design Recovery [**Intersolv91**] which translates code into diagrams which clarify the underlying structure. It reads COBOL source code and database definitions, then generates the corresponding physical models, which it stores in an internal repository. The models may be examined and changed, and then used to produce new code.

Design Recovery uses diagrams to show the hierarchical relationship between sections and paragraphs. It automatically identifies dead code. For every section, it maintains a list of which variables and which files are updated. It also calculates a metric measuring the complexity of every section.

In the description of all these products, the use of the term *re-engineering* is quite liberal [**Chiko90**]. These products focus on presenting the code in an attractive form, and supporting the user by giving him the power to easily navigate through the code. Some of them help in organizing documentation, and/or testing and debugging. A good source of information in this area is [**Ovum90**].

The bottom line is that there are a number of products which are capable of analyzing existing code (and sometimes data), and presenting it to the user in a more attractive form than that provided by a simple text editor. There is virtually no mention of automatic regeneration of programs.

5.2. Tools in Research Labs. A number of research centers have allocated resources to the software maintenance crisis. It has been realized that the most time-consuming aspect of maintenance is program understanding. Therefore, any system which can automate this process will greatly increase productivity.

Ourston [**Ourston89**] looked at a number of research projects which addressed this problem. Program recognition is a form of program analysis that identifies the purpose of a program. It is necessary for software maintenance, where programmers must modify code which is unfamiliar to them, and which has possibly not been developed using acceptable software engineering techniques.

The Program Recognizer [**Wills92**] uses a library of clichés to identify fragments and data structures which appear in the code. It is possible that there will be gaps in the understanding of the program, but this is acceptable as long as individual clichés can still be recognized. The order or separation of statements in the input program will not affect recognition. Program Recognizer requires that all pertinent clichés exist in the database before the recognition process begins.

Talus [**Brotsky84**] was developed to provide automatic program debugging in support of intelligent tutoring. It reads in and attempts to correct errors in LISP programs, by comparing them with correct versions of the program.

Proust [**Johnson85**] is part of an intelligent tutoring system for novice programming students. It tries to debug Pascal programs by identifying programmer intentions. Proust uses a top-down template matching approach which minimizes the search space required for successful program identification. By operating with goals at each step, it can synthesize solutions which do not appear in the database. Proust requires that the problem to solve be specified.

Rigi [**Muller91**] was initially developed as a tool for programming-in-the-large, but it has been expanded and is now also a maintenance and re-engineering tool. The first step is to parse

the target program and store its artifacts in a repository. This is done by extracting of relevant system components and dependencies out of the source code. The second step, which requires interactive assistance from the programmer, is the generation of hierarchies of subsystems, based on the resource-flow graphs of the source code. The third step is the construct of interfaces among the subsystems. The final step is the evaluation of the subsystem structures using established software engineering principles as a guide. Reverse engineering can be used to produce documentation which is more consistent and accurate.

Design recovery [**Biggerstaff94**] is needed primarily for maintenance, enhancement and reuse. A domain model is used to store expertise about the program in question. The goal is to develop structures which will aid in understanding a program. Expectations drawn from previous experience serve to guide the understanding process. The first step is to identify the most important modules and data structures. A conceptual abstraction is associated with each module, and a mapping is maintained to its corresponding code. Conceptual abstractions are used to hide the detail inherit in a complex module, allowing one to focus on the code in terms of levels of abstraction. The software engineer has knowledge of numerous conceptual abstractions and seeks instances of these in the code. The informal clues provided by variable and function names is vital to program comprehension. Programming languages do not contain the constructs necessary to express information about the informal conceptual abstractions behind the code. A system called Desire has been implemented and has been used to extract design knowledge to support translating C to C++. The importance of domain knowledge, as well as the intuition of an experienced programmer, is stressed in this approach.

Ward, Calliss and Munro [**Ward89**] have developed "The Maintainer's Assistant" which provides a structured framework wherein maintenance can be approached. The largest part of the maintenance task is the understanding of the code; working out what the program is supposed to do, and how sections of code affect one another. Understanding a program involves several aspects:

- understanding the specifications of its modules,
- understanding the data flow,
- understanding the control flow,
- determining the scope of the variables, and
- determining the effects of a proposed modification.

A number of different kinds of transformations are used:

- local and global restructuring of code,
- expression in a higher-level notation, and
- restructuring of data structures.

The system which has been developed is an interactive one, as not enough information is present in the code itself to make assumptions about the function of the program and its various parts. When viewing the program through the browser, the user may specify that a section of code should be reduced to its specification, and vice versa. The system consists of a program transformer which uses a knowledge base to convert a section of code to equivalent but more structured code, and a structure editor, which is a syntax-based editor through which all changes must be made.

Canfora, and Cimitile [**Canfora94**], [**Canfora92**], [**Cimitile90**] present efficient algorithms for analyzing the control and data flow in order to identify binding conditions on program variable's. Parts of the system that are bound by a set of conditions are isolated. This approach combines formal representation methods (e.g. Control Flow Graph, Program Dependency Graph) with first order logic to provide horizontal and vertical slices of a Program. This approach is used to identify components that may implement a particular function or lie on the same dynamic path.

Baker [**Baker94**], [**Baker95**] represents source code a stream of strings. The approach uses parameterized Pattern Matching techniques based on a variation of a variation of the Boyer-Moore algorithm to identify duplication within a string. A prototype system called *dup* has been implemented and is currently used in large legacy systems. The strength of this approach is the efficient matching and the overall scalability of the algorithm used. The drawback is that it does not relate to any program feature such as the control and data flow.

Paul [**Paul94**] proposes a system (SCRUPLE) in which regular-expressions are used to locate programming patterns in a large software system. Pattern matching is performed by testing if a code fragment is accepted by the automaton that is constructed by a regular-expression provided by the user, as a query. The advantage of this approach is that it is string based, may be easily applied to a variety of programming languages, and has been demonstrated to be fast. The limitations for this approach are that it allows for exact matching only and does not provide any means for modeling abstractions or allow for hierarchical recognition.

Johnson [**Johnson94a**], [**Johnson94b**] uses a similar text based approach where fingerprints in source files are computed using a hashing mechanism. Fingerprints are compared to identify an overall similarity between two texts. The advantage of this approach is that it is very fast, scalable and efficient. Its drawback is that it allows only for exact matching, and it may produce noise in the recognition process by matching text irrelevant to the source code (e.g headers, include files etc.)

Jankowitz [**Jankowitz88**], and McCabe [**McCabe90**] use statistical measurements to compute a fingerprint of a software component. This is close to our approach using the metrics-based matching. The significant difference of our approach form these methods is that we compute all measurements compositionally at the AST nodes and we provide a methodology of segmenting and

delineating the source code in order to achieve matching at a granularity lower than a *a Function* or a *Procedure*.

CHAPTER 3

Program Features For Design Recovery

1. Introduction

In order to perform design recovery we have first to represent source code in a higher level of abstraction. The program representation scheme must allow for the calculation of a number of feature vectors and be able to to do so for every statement, block or function of the source code.

For our program representation we use an object-oriented annotated abstract syntax tree (AST). We have chosen this program representation scheme because:

- it does not require any overhead to be computed as it is a direct product of the parsing process and,
- it can be easily analyzed to compute several data and control flow program properties

Nodes of the AST are represented as objects in a LISP-based development environment[1]. A sample AST for a C code fragment in Fig.3.1, where the AST, its root in its Object representation, and the corresponding source code are illustrated.

Creating the annotated AST is a three-step process. First, a grammar and an object (domain) model must be designed for the programming language of the subject system. The tool vendor has parsers available for such common languages as C and COBOL. Parsers for other languages may be easily constructed or obtained through the user community. The domain model defines object-oriented hierarchies for the AST nodes in which, for example, an *If-Statement* and a *While-Statement* are defined to be subclasses of the *Statement* class.

The second step is to use the parser on the subject system to construct the AST representation of the source code. Some tree annotations, such as linkage information and the call graph are created automatically by the parser. Other tree annotations are computed as part of the software

[1] We are using as our development environment a commercial tool called REFINE (a trademark of Reasoning Systems Corp.).

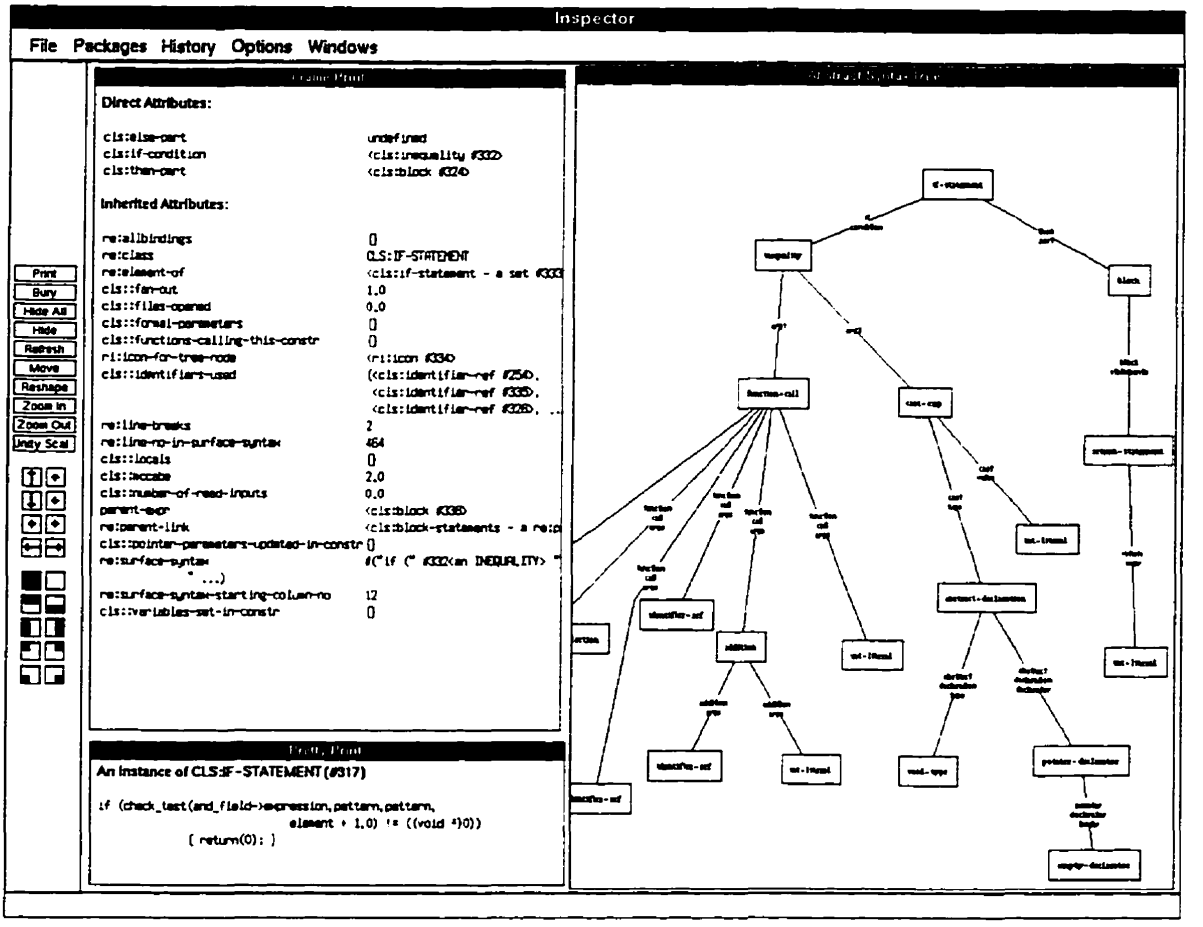


FIGURE 3.1. AST nodes are represented as objects in a local repository. Arcs of the AST are represented as mappings between objects.

analysis process (i.e. metrics). Once the AST is created, further steps operate in an essentially language-independent fashion.

The final step is to add additional annotations into the tree for information on data types, data flow (data flow graphs), the results of external analysis, and links to informal information. An important aspect proposed in this thesis is that we calculate all these annotations *compositionally* from the leaves to the root of the AST, that is from *Expressions* to *Statements* to *Blocks*, and finally to *Functions*.

This chapter discusses the nature of annotations that we added to the AST in order to facilitate pattern matching and perform plan localization.

2. Program Feature Vectors for Clone Detection

In this section we describe the features used for classifying code fragments, and discuss the way that have been compositionally computed for all *Expressions*, *Statements*, *Blocks*, and finally *Functions* in the system.

The program features were selected based on their contribution to the data and control flow of the system. We aimed for the features to exhibit low correlation (based on the Spearman-Pierson correlation test) so as to be sensitive to different control and data flow properties and each one to contribute an independent program characteristic. The features selected for our analyses include:

(i) The number of functions called (Fanout);

(ii) Global and local variables [2] used and updated;

(iii) Parameters passed by reference used and updated;

(iv) Parameters passed by value used and updated;

(v) Input/Output operations;

(vi) External files used;

(vii) McCabe cyclomatic complexity;

(viii) Albrecht's function point metric;

(ix) Henry-Kafura's information flow quality metric

For example, consider the following code fragment from a proprietary PL/1-like language.

```
MAIN: PROCEDURE(OPTION);
      DCL OPTION FIXED(31);
      IF (OPTION>0) THEN
          CALL SHOW_MENU(OPTION);
      ELSE
          CALL SHOW_ERROR("Invalid option number");
END MAIN;
```

The corresponding AST representation for the IF statement is shown in Fig. 3.2. The tree is annotated with the *Fanout* attribute which has been determined during an analysis phase following the initial parse.

The annotation has been computed in a *compositional* way. The final *Fanout* value for the IF statement has been calculated in terms of the *Fanout* of its THEN and ELSE parts, which in turn obtained their values by composing the values from the nodes below them. Compositionality is a very important aspect of this approach as it allows to classification of code entities irrespective of their class. In such a way a WHILE statement can be compared with a Block statement and found

[2] Variables are also referred in the text as Identifiers

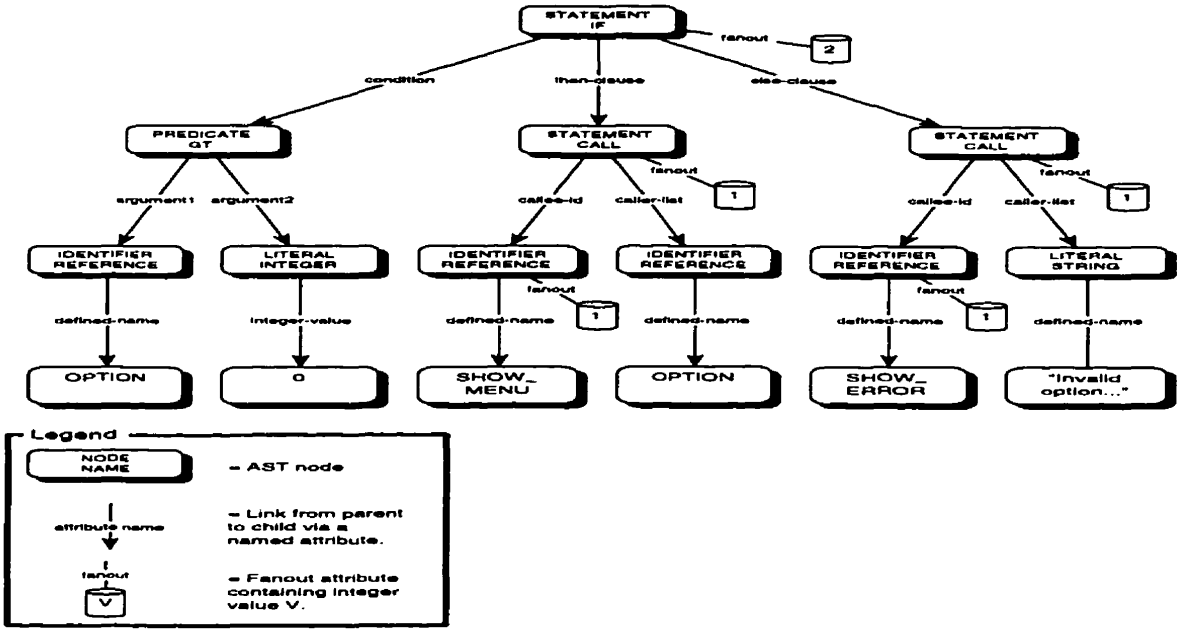


FIGURE 3.2. The AST for an IF statement with Fanout attributes.

similar if their corresponding feature vectors match. A typical scenario is when a cut-and-paste operation takes this `While` statement and inserts it in a `Block` in another part of the system. In the following sections, the features used for the proposed pattern matching techniques are presented in detail.

2.1. Global Variables.

- Description : GLOBALS(a_constr) is the set of global variables used or updated within the construct *a_constr*.

  A global variable for a Statement or Expression or Function is a variable which is not declared[3] in the Statement, the Expression or the Function.

- Cases :

  (i) If **a_constr is a FUNCTION** then

$$GLOBALS(a\_constr) = \{\ GLOBALS(Body)$$

  where *Body* is the body of the function *a_constr*

[3]A variable declaration is a point where the variable is formally declared. A variable definition is a point where a variable is stored/updated.

(ii) If **a_constr is a sequence of statements** $S_1 \ldots S_n$ then

$$GLOBALS(a\_constr) = \{ \cap_{i=1}^{n} GLOBALS(S_i)$$

(iii) If **a_constr is an IF statement** then

$$GLOBALS(a\_constr) = \begin{cases} GLOBALS(Cond) \cap \\ GLOBALS(ThenPart) \cap \\ GLOBALS(ElsePart) \end{cases}$$

where *Cond, ThenPart, ElsePart* are the condition the Then part and the Else part of the IF statement *a-constr* respectively

(iv) If **a_constr is a WHILE statement** then

$$GLOBALS(a\_constr) = \begin{cases} GLOBALS(Cond) \cap \\ GLOBALS(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(v) If **a_constr is a DO statement** then

$$GLOBALS(a\_constr) = \begin{cases} GLOBALS(Cond) \cap \\ GLOBALS(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the DO statement *a_constr* respectively

(vi) If **a_constr is a FOR statement** then

$$GLOBALS(a\_constr) = \begin{cases} GLOBALS(Init) \cap \\ GLOBALS(Incr) \cap \\ GLOBALS(Test) \cap \\ GLOBALS(Body) \end{cases}$$

where *Init, Incr, Test, Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vii) If **a_constr is a GOTO statement** then

$$GLOBALS(a\_constr) = \{ GLOBALS(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(viii) If **a_constr is a SWITCH statement** then

$$GLOBALS(a\_constr) = \begin{cases} GLOBALS(Cond) \cap \\ GLOBALS(Body) \end{cases}$$

where *Cond, Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(ix) If **a_constr is a RETURN statement** then

$$GLOBALS(a\_constr) = \{ \; GLOBALS(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(x) If **a_constr is a LABELED statement** then

$$GLOBALS(a\_constr) = \{ \; GLOBALS(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(xi) If **a_constr is an EXPRESSION Statement** (e.g. an assignment) then

$GLOBALS(a\_constr) = \{$ the number of individual variables used or updated within *a_constr* and not declared within a_constr

(xii) If **a_constr is an EXPRESSION** then

$GLOBALS(a\_constr) = \{$ the number of individual variables used or updated within *a_constr* and not declared within a_constr

2.2. Global Variables Updated.

- Description : GLOBALS_UPDATED(a_constr) is equal to the set of global variables updated within the construct *a_constr.*
- Cases :

(i) If **a_constr is a FUNCTION** then

$$GLOBALS\_UPDATED(a\_constr) = \{ \; GLOBALS\_UPDATED(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** $S_1 \ldots S_n$ then

$$GLOBALS\_UPDATED(a\_constr) = \{ \cap_{i=1}^{n} GLOBALS\_UPDATED(S_i)$$

(iii) If **a_constr is an IF statement** then

$$GLOBALS\_UPDATED(a\_constr) = \begin{cases} GLOBALS\_UPDATED(Cond) \cap \\ GLOBALS\_UPDATED(ThenPart) \cap \\ GLOBALS\_UPDATED(ElsePart) \end{cases}$$

where *Cond, ThenPart, ElsePart* are the condition the Then part and the Else part of the IF statement *a-constr* respectively

(iv) If **a_constr is a WHILE statement** then

$$GLOBALS\_UPDATED(a\_constr) = \begin{cases} GLOBALS\_UPDATED(Cond) \cap \\ GLOBALS\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(v) If **a_constr is a DO statement** then

$$GLOBALS\_UPDATED(a\_constr) = \begin{cases} GLOBALS\_UPDATED(Cond) \cap \\ GLOBALS\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the DO statement *a_constr* respectively

(vi) If **a_constr is a FOR statement** then

$$GLOBALS\_UPDATED(a\_constr) = \begin{cases} GLOBALS\_UPDATED(Init) \cap \\ GLOBALS\_UPDATED(Incr) \cap \\ GLOBALS\_UPDATED(Test) \cap \\ GLOBALS\_UPDATED(Body) \end{cases}$$

where *Init, Incr, Test, Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vii) If **a_constr is a GOTO statement** then

$$GLOBALS\_UPDATED(a\_constr) = \{ GLOBALS\_UPDATED(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(viii) If **a_constr is a SWITCH statement** then

$$GLOBALS\_UPDATED(a\_constr) = \begin{cases} GLOBALS\_UPDATED(Cond) \cap \\ GLOBALS\_UPDATED(Body) \end{cases}$$

where *Cond*, *Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(ix) If **a_constr is a RETURN statement** then

$$GLOBALS\_UPDATED(a\_constr) = \{\ GLOBALS\_UPDATED(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(x) If **a_constr is a LABELED statement** then

$$GLOBALS\_UPDATED(a\_constr) = \{\ GLOBALS\_UPDATED(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(xi) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

$GLOBALS\_UPDATED(a\_constr) = \{$ the set of individual variables updated
within a_constr and
not declared within a_constr

(xii) If **a_constr is an EXPRESSION** then

$GLOBALS\_UPDATED(a\_constr) = \{$ the set of individual variables updated
within a_constr and
not declared within a_constr[4]

2.3. Input / Output.

- Description : READ_STATS(a_constr) is equal to the set of input statements in the construct *a_constr*. In the case of C these are: sscanf, scanf, fscanf, getc, getchar, gets, fgetc, and fgets.

- Cases :

[4]The updates are based on Assignments, Post/Pre Incrementation, and Post/Pre Decrementation Statements

(i) If **a_constr is a FUNCTION** then

$$READ\_STATS(a\_constr) = \{\ READ\_STATS(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** S_1 ... S_n then

$$READ\_STATS(a\_constr) = \{\cup_{i=1}^{n} READ\_STATS(S_i)$$

(iii) If **a_constr is a WHILE statement** then

$$READ\_STATS(a\_constr) = \begin{cases} READ\_STATS(Cond)\cup \\ READ\_STATS(Body) \end{cases}$$

where *Cond*, *Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(iv) If **a_constr is a DO statement** then

$$READ\_STATS(a\_constr) = \begin{cases} READ\_STATS(Cond)\cup \\ READ\_STATS(Body) \end{cases}$$

where *Cond*, *Body* are the condition and the Body of the DO statement *a_constr* respectively

(v) If **a_constr is a FOR statement** then

$$READ\_STATS(a\_constr) = \begin{cases} READ\_STATS(Init)\cup \\ READ\_STATS(Incr)\cup \\ READ\_STATS(Test)\cup \\ READ\_STATS(Body) \end{cases}$$

where *Init*, *Incr*, *Test*, *Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vi) If **a_constr is a GOTO statement** then

$$READ\_STATS(a\_constr) = \{\ READ\_STATS(Label)$$

where

Label is the Label expression of the GOTO statement *a_constr*

(vii) If **a_constr is a SWITCH statement** then

$$READ\_STATS(a\_constr) = \begin{cases} READ\_STATS(Cond) \cup \\ READ\_STATS(Body) \end{cases}$$

where *Cond, Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(viii) If **a_constr is a RETURN statement** then

$$READ\_STATS(a\_constr) = \{\ READ\_STATS(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(ix) If **a_constr is a LABELED statement** then

$$READ\_STATS(a\_constr) = \{\ READ\_STATS(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(x) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

READ_STATS(a_constr) = the set of Input related function calls in the construct

(xi) If **a_constr is an EXPRESSION** then

READ_STATS(a_constr) = the set of Input related function calls in the construct

2.4. Files Opened.

- Description : FILES_OPENED(a_constr) is equal to the number of files opened in the construct *a_constr*. In the of programs written in C the number of files opened is equal to the number of *fopen* function calls

- Cases :

  (i) If **a_constr is a FUNCTION** then

$$FILES\_OPENED(a\_constr) = \{\ FILES\_OPENED(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** S_1 ... S_n then

$$FILES\_OPENED(a\_constr) = \{\sum_{i=1}^{n} FILES\_OPENED(S_i)$$

(iii) If **a_constr is a WHILE statement** then

$$FILES\_OPENED(a\_constr) = \begin{cases} FILES\_OPENED(Cond)+ \\ FILES\_OPENED(Body) \end{cases}$$

where *Cond*, *Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(iv) If **a_constr is a DO statement** then

$$FILES\_OPENED(a\_constr) = \begin{cases} FILES\_OPENED(Cond)+ \\ FILES\_OPENED(Body) \end{cases}$$

where *Cond*, *Body* are the condition and the Body of the DO statement *a_constr* respectively

(v) If **a_constr is a FOR statement** then

$$FILES\_OPENED(a\_constr) = \begin{cases} FILES\_OPENED(Init)+ \\ FILES\_OPENED(Incr)+ \\ FILES\_OPENED(Test)+ \\ FILES\_OPENED(Body) \end{cases}$$

where *Init*, *Incr*, *Test*, *Body* are the Initialize expression, the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vi) If **a_constr is a GOTO statement** then

$$FILES\_OPENED(a\_constr) = \{ FILES\_OPENED(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(vii) If **a_constr is a SWITCH statement** then

$$FILES\_OPENED(a\_constr) = \begin{cases} FILES\_OPENED(Cond)+ \\ FILES\_OPENED(Body) \end{cases}$$

where *Cond*, *Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(viii) If **a_constr is a RETURN statement** then

$$FILES\_OPENED(a\_constr) = \{\ FILES\_OPENED(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(ix) If **a_constr is a LABELED statement** then

$$FILES\_OPENED(a\_constr) = \{\ FILES\_OPENED(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(x) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

FILES_OPENED(a_constr) = the number of fopen function calls in the construct

(xi) If **a_constr is an EXPRESSION** then

FILES_OPENED(a_constr) = the number of fopen function calls in the construct

2.5. Formal Parameters.

- Description : FORMAL_PARMS(a_constr) is equal to the set of formal parameters of *a_constr* (applies when *a_constr* is a *Function*).
- Cases :
  (i) If **a_constr is a Function** then
  $FORMAL\_PARMS(a\_constr)$ = the formal parameter list of *a_constr*.
  (ii) If **a_constr is any other construct** then $FORMAL\_PARMS(a\_constr) = \emptyset$

2.6. Parameters by Reference Updated.

- Description : PARMS_BY_REF_UPDATED(a_constr) is equal to the set of pointer variables declared in the formal parameter list of the containing function and are updated within the construct *a_constr*
- Cases :
  (i) If **a_constr is a FUNCTION** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \{\ PARMS\_BY\_REF\_UPDATED(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** $S_1 \ldots S_n$ then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \{\cup_{i=1}^{n} PARMS\_BY\_REF\_UPDATED(S_i)$$

(iii) If **a_constr is a WHILE statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \begin{cases} PARMS\_BY\_REF\_UPDATED(Cond) \cup \\ PARMS\_BY\_REF\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(iv) If **a_constr is a DO statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \begin{cases} PARMS\_BY\_REF\_UPDATED(Cond) \cup \\ PARMS\_BY\_REF\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the DO statement *a_constr* respectively

(v) If **a_constr is a FOR statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \begin{cases} PARMS\_BY\_REF\_UPDATED(Init) \cup \\ PARMS\_BY\_REF\_UPDATED(Incr) \cup \\ PARMS\_BY\_REF\_UPDATED(Test) \cup \\ PARMS\_BY\_REF\_UPDATED(Body) \end{cases}$$

where *Init, Incr, Test, Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vi) If **a_constr is a GOTO statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \{ PARMS\_BY\_REF\_UPDATED(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(vii) If **a_constr is a SWITCH statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \begin{cases} PARMS\_BY\_REF\_UPDATED(Cond) \cup \\ PARMS\_BY\_REF\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(viii) If **a_constr is a RETURN statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \{\ PARMS\_BY\_REF\_UPDATED(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(ix) If **a_constr is a LABELED statement** then

$$PARMS\_BY\_REF\_UPDATED(a\_constr) = \{\ PARMS\_BY\_REF\_UPDATED(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(x) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

$PARMS\_BY\_REF\_UPDATED(a\_constr)$ = set of pointer variables declared in the formal parameter list of the containing function and are updated within *a_constr*

(xi) If **a_constr is an EXPRESSION** then

$PARMS\_BY\_REF\_UPDATED(a\_constr)$ = set of pointer variables declared in the formal parameter list of the containing function and are updated within *a_constr*

2.7. Identifiers Used.

- Description : IDS_USED(a_constr) is the set of variables used in the construct *a_constr*
- Cases :

(i) If **a_constr is a FUNCTION** then

$$IDS\_USED(a\_constr) = \{\ IDS\_USED(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** S_1 ... S_n then

$$IDS\_USED(a\_constr) = \{\cup_{i=1}^{n} IDS\_USED(S_i)$$

(iii) If **a_constr is a WHILE statement** then

$$IDS\_USED(a\_constr) = \begin{cases} IDS\_USED(Cond) \cup \\ IDS\_USED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(iv) If **a_constr is a DO statement** then

$$IDS\_USED(a\_constr) = \begin{cases} IDS\_USED(Cond) \cup \\ IDS\_USED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the DO statement *a_constr* respectively

(v) If **a_constr is a FOR statement** then

$$IDS\_USED(a\_constr) = \begin{cases} IDS\_USED(Init) \cup \\ IDS\_USED(Incr) \cup \\ IDS\_USED(Test) \cup \\ IDS\_USED(Body) \end{cases}$$

where *Init, Incr, Test, Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vi) If **a_constr is a GOTO statement** then

$$IDS\_USED(a\_constr) = \{ IDS\_USED(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(vii) If **a_constr is a SWITCH statement** then

$$IDS\_USED(a\_constr) = \begin{cases} IDS\_USED(Cond) \cup \\ IDS\_USED(Body) \end{cases}$$

where *Cond, Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(viii) If **a_constr is a RETURN statement** then

$$IDS\_USED(a\_constr) = \{ IDS\_USED(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(ix) If **a_constr is a LABELED statement** then

$$IDS\_USED(a\_constr) = \{\ IDS\_USED(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(x) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

$IDS\_USED(a\_constr)$ = the set of variables used in the construct *a_constr*

(xi) If **a_constr is an EXPRESSION** then

$IDS\_USED(a\_constr)$ = the set of variables used in the construct *a_constr*

2.8. Identifiers Updated.

- Description : IDS_UPDATED(a_constr) is the set of variables updated in the construct *a_constr*
- Cases :

(i) If **a_constr is a FUNCTION** then

$$IDS\_UPDATED(a\_constr) = \{\ IDS\_UPDATED(Body)$$

where *Body* is the body of the function *a_constr*

(ii) If **a_constr is a sequence of statements** S_1 ... S_n then

$$IDS\_UPDATED(a\_constr) = \{\cup_{i=1}^{n} IDS\_UPDATED(S_i)$$

(iii) If **a_constr is a WHILE statement** then

$$IDS\_UPDATED(a\_constr) = \begin{cases} IDS\_UPDATED(Cond)\cup \\ IDS\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the WHILE statement *a_constr* respectively

(iv) If **a_constr is a DO statement** then

$$IDS\_UPDATED(a\_constr) = \begin{cases} IDS\_UPDATED(Cond)\cup \\ IDS\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the condition and the Body of the DO statement *a_constr* respectively

(v) If **a_constr is a FOR statement** then

$$IDS\_UPDATED(a\_constr) = \begin{cases} IDS\_UPDATED(Init) \cup \\ IDS\_UPDATED(Incr) \cup \\ IDS\_UPDATED(Test) \cup \\ IDS\_UPDATED(Body) \end{cases}$$

where *Init, Incr, Test, Body* are the Initialize expression the Increment expression, the Test expression and the Body of the FOR statement *a_constr* respectively

(vi) If **a_constr is a GOTO statement** then

$$IDS\_UPDATED(a\_constr) = \{\ IDS\_UPDATED(Label)$$

where *Label* is the Label expression of the GOTO statement *a_constr*

(vii) If **a_constr is a SWITCH statement** then

$$IDS\_UPDATED(a\_constr) = \begin{cases} IDS\_UPDATED(Cond) \cup \\ IDS\_UPDATED(Body) \end{cases}$$

where *Cond, Body* are the Switch Test expression and Body of the SWITCH statement *a_constr* respectively

(viii) If **a_constr is a RETURN statement** then

$$IDS\_UPDATED(a\_constr) = \{\ IDS\_UPDATED(return\_expr)$$

where *return_expr* is the return expression of the RETURN statement *a_constr*

(ix) If **a_constr is a LABELED statement** then

$$IDS\_UPDATED(a\_constr) = \{\ IDS\_UPDATED(Body)$$

where *Body* is the Body of the LABELED statement *a_constr*

(x) If **a_constr is an EXPRESSION STATEMENT** (e.g. an assignment) then

$IDS\_UPDATED(a\_constr)$ = the set of variables updated in the construct

(xi) If **a_constr is an EXPRESSION** then

$IDS\_UPDATED(a\_constr)$ = the set of variables updated in the construct

2.9. Function Calls.

- Description : FUNCTION_CALLS_TO(a_constr) is equal to the set of individual of Function Calls to *a_constr* (applies only when *a_constr* is a *Function*)
- Cases :

  (i) If **a_constr is a Function** then

  $$FUNCTION\_CALLS\_TO(a\_constr) = \text{set of individual function calls to } a\_constr$$

  (ii) If **a_constr is any other construct** then

  $$FUNCTION\_CALLS\_TO(a\_constr) = \emptyset$$

2.10. S-Complexity.

- Description : S_COMPLEXITY(a_constr) is equal to $|FAN\_OUT(a\_constr)|^2$ where $|FAN\_OUT(a\_constr)|$ is the number of individual function calls in the construct (a_constr)

2.11. D-Complexity.

- Description :

$$D\_COMPLEXITY(a\_constr) = |GLOBALS(a\_constr)|/(|FAN\_OUT(a\_constr)| + 1)$$

where $|GLOBALS(a\_constr)|$ is the number of individual declarations of global variables used or updated within the construct *a_constr*. A global variable for a Statement or Expression or Function is a variable which is not declared in the Statement, the Expression or the Function.

2.12. McCabe Complexity. MCCABE

- Description : MCCABE(a_constr) is equal to $\epsilon - n + 2$
  where ϵ is the number of edges in the control flow graph of the construct *a_constr* and n is the number of nodes in the same graph.
  Alternatively McCabe metric can be calculated as :
  MCCABE(a_constr) = 1 + d where d is the number of control decision predicates in the construct *a_constr*
- Cases :

(i) If **a_constr is a simple statement** then

$$MCCABE(a\_constr) = \{\ 1 + d$$

where d is the number of control decision predicates in the construct *a_constr*

(ii) If **a_constr is a sequence of statements** $S_1, ..S_k$ then

$$MCCABE(a\_constr) = \{\ \sum_{i=1}^{k} MCCABE(S_i) - k + 1$$

(iii) If **a_constr is a composite statement of k statements** then

$$MCCABE(a\_constr) = \begin{cases} MCCABE(S)+ \\ \sum_{i=1}^{k} MCCABE(S_i) - k \end{cases}$$

where S is the statement *a_constr* viewed as a simple statement

(iv) If **a_constr is an Expression** then

$$MCCABE(a\_constr) = 1$$

2.13. Albrecht Metric.

- Description :

$$ALBRECHT(a\_constr) = \begin{cases} p_1 * |GLOBALS(a\_constr)|+ \\ p_2 * (|GLOBALS\_UPDATED(a\_constr)|+ \\ \qquad |PARMS\_BY\_REF\_UPDATED(a\_constr)|)+ \\ p_3 * |READ\_STATS(a\_constr)|+ \\ p_4 * FILES\_OPENED(a\_constr) \end{cases}$$

where

- $|GLOBALS(a\_constr)|$ is the is the number of individual declarations of global variables used or updated within the construct *a_constr*.

- $|GLOBALS\_UPDATED(a\_constr)|$ is the number of individual declarations of global variables updated within the construct *a_constr*.

- $|PARMS\_BY\_REF\_UPDATED(a\_constr)|$ is the number of pointer type variables in the formal parameter list of the Function in which a_constr is contained and which

variables are updated within the construct *a_constr* [5].

- $|READ\_STATS(a\_constr)|$ is the number of input statements in the construct *a_constr*. These statements include the C statements : sscanf, scanf, fscanf, getc, getchar, gets, fgetc, and fgets.

- FILES_OPENED(a_constr) is the number of fopen statements in the construct a_constr.

- The parameters p_i have integer values. The current implementation uses the following values[**Adamov87**] :

  $p_1 = 4$

  $p_2 = 5$

  $p_3 = 4$

  $p_4 = 7$

2.14. Kafura Metric.

- Description :

$$KAFURA(a\_constr) = \left\{ \; (KAFURA\_IN(a\_constr) * KAFURA\_OUT(a\_constr))^2 \right.$$

where

(i) KAFURA_IN(a_constr) is the sum of

  (a) the number of formal parameters ($|FORMAL\_PARMS(a\_constr)|$)

  (b) the number of variables ($|IDS\_USED(a\_constr)|$) used in the construct *a_constr*,

  (c) the number of Function Calls to *a_constr* ($|FUNCTION\_CALLS\_TO(a\_constr)|$)

(ii) KAFURA_OUT(a_constr) is the sum of

  (a) number of Functions called by *a_constr* (that is the same as $|FAN\_OUT(a\_constr)|$),

  (b) the number of individual declarations of global variables updated within the construct *a_constr* (that is $|GLOBALS\_UPDATED(a\_constr)|$),

  (c) the number of pointer type variables in the formal parameter list of the function in which a_constr is contained and which variables are updated within the construct *a_constr*, that is $|PARMS\_BY\_REF\_UPDATED(a\_constr)|$.

[5] Updates are calculated based on Assignment, Pre/Post Incrementation, and Pre/Post Decrementation statements

3. Pattern Matching

In the following two chapters we discuss the proposed pattern-matching algorithms applied to the problem of clone detection, and plan recognition. Determining whether two arbitrary program functions have identical behavior is known to be undecidable in the general case. Our approach to clone detection exploits the observation that clone instances, by their nature, should have a high degree of structural similarity and data flow similarity. We look for identifiable characteristics or features that can be used as a signature to categorize arbitrary pieces of code.

The work presented here uses *feature vectors* to establish similarity measures. Features examined include metric values and specific data- and control-flow properties. In this thesis, we present the following three types of pattern matching techniques for code cloning detection and plan localization:

(i) metric-value similarity analysis,

(ii) dynamic programming techniques for comparing two code fragments at a statement-by-statement basis and,

(iii) stochastic matching based on Dynamic Programming and Markov Models that represent formulations of abstract descriptions of programming plans

Metric-value similarity analysis is based on the assumption that two code fragments C_1 and C_2 have metric values $M(C_1)$ and $M(C_2)$ for some source code metric M. If the two fragments are similar under the set of features measured by M, then the values of $M(C_1)$ and $M(C_2)$ should be proximate.

Dynamic Programming based similarity analysis performs comparisons of program features at a statement-by-statement basis. A Dynamic Programming function allows for calculating the best fit between two sequences of program statements. The best fit is calculated by comparing program features instead of just text. Within the Dynamic Programming framework we have experimented with the following program features at per statement level:

(i) Sets and Uses of Variables

(ii) Sets and Uses of Data Types

(iii) Metrics (as discussed in Section.2)

Finally, stochastic matching is based on a pattern language that describes in a high level of abstraction the structure and the several data flow properties of a code fragment that may represent a particular algorithmic plan. A matching algorithm is used to localize and match code fragments that may be generated (matched) by this abstract description.

These experiments were conducted within the framework of the Program Understanding project with IBM Canada, Center for Advanced Studies.

CHAPTER 4

Code To Code Matching

1. Metric-Value Similarity Analysis

Metric-value similarity analysis is based on the assumption that if two code fragments have similar metrics then may have similar structure, data flow and control flow characteristics. These metrics have been chosen so that they represent and classify a number of low correlated program features that are sensitive to program structure, I/O patterns, as well as data flow and control flow properties.

Within this framework, the calculation of similarity between two code fragments becomes a matter of comparing features as these are represented by five software metrics.

The Five modified metrics [**Adamov87**], [**Fenton91**], [**Buss94**] discussed above for which their components exhibit low correlation (based on the Spearman-Pierson correlation test)[**Buss94**] were selected for our analyses. These selected metrics are:

(i) The number of functions called (fanout);
(ii) The ratio of input/output variables to the fanout;
(iii) McCabe cyclomatic complexity;
(iv) Albrecht's function point metric;
(v) Henry-Kafura's information flow quality metric.

Once the five metrics M_1 to M_5 are computed for every statement, block and function node, the pattern matching process is very fast and efficient because it is based on the comparison of numeric values.

We have experimented with two techniques for calculating the similarity of code fragments in a software system.

The first one is based on pairwise Euclidean distance comparison of all statements that are of length more than n lines long, where n is a parameter given by the user. We refer to it as *Partitioning Clustering Similarity Analysis* because it is based on a partition clustering algorithm using Euclidean distances to compute similarity between two entities.

The second technique is more efficient and uses an hierarchical clustering algorithm applied in sequence to all five metric dimensions, and we refer to it as *Hierarchical Clustering Similarity Analysis.*

The issues related to Metric-Value Similarity analysis are:

(i) code delineation criteria and selection of level of granularity for the matching process.

(ii) selection of a comparison function to calculate distances between two metric vectors.

(iii) selection of appropriate threshold values per metric dimension by which two code fragments can be considered similar. The selection of an appropriate threshold value for a metric dimension is based on the nature of the metric and the tolerance the user is willing to accommodate towards partial matching.

These points are discussed in more detail in the following sections.

1.1. Hierarchical Clustering Clone Detection. A hierarchical clustering method is a procedure for transforming a proximity matrix into a sequence of nested partitions [**Jain88**]. A proximity matrix is a matrix for which each element in $d_{i,j}$ denotes the distance between two elements or patterns X_i, X_j. Hierarchical clustering as well as Partitioning clustering (discussed in the following section) is applied to *exclusive* and *intristic* classifications. An *exclusive* classification is a partition of a set of objects in which each object belongs exactly to one cluster. An *intristic* or *unsupervised learning* classification is a partition that uses only the proximity matrix to compute it.

In literature [**Clifford75**], [**Shepard79**], [**Sneath73**], [**Jain88**] the primary algorithmic options for performing clustering are presented. In brief, these are classified as, *agglomerative versus divisive, serial versus simultaneieous, monothetic versus polythetic.*

An *agglomerative*, hierarchical clustering algorithm starts by placing each object in its own cluster and continues by merging into larger and larger clusters. A *serial* algorithm handles the patterns one by one, while a *monothetic* algorithm uses the classification features one by one.

We propose an exclusive, intristic, agglomerative, serial, and monothetic algorithm for computing a partition that contains clusters of code fragments that are considered similar within their cluster.

The technique starts by creating clusters of potential clones for every metric axis $\mathcal{M}_i$ (i = 1 .. 5). Once the clusters for each axis are created, then intersections of clusters in different axes are

calculated forming intermediate results. For example, every cluster in the axis $\mathcal{M}_i$ contains potential clones under the criteria implied by this metric. Consequently, every cluster that has been calculated by intersecting clusters in $\mathcal{M}_i$ and $\mathcal{M}_j$ contains potential clones under the criteria implied by *both* metrics. The process ends when all metric axes have been considered. The user may specify at the beginning the order of comparison, and the clustering thresholds for every metric axis. The clone detection algorithm that is using clustering can be summarized as:

- **Step 1.** Select all source code statements S from the AST that are more than n lines long. The parameter n can be changed by the user.
- **Step 2.** For each metric axis $\mathcal{M}_i$ (i = 1 .. 5) create clusters $C_{i,j}$ that contain statements with distance less than a given threshold d_i that is selected by the user. Each cluster then contains potential code clone fragments under the metric criterion $\mathcal{M}_i$. Set the current axis $\mathcal{M}_{curr} = \mathcal{M}_i$, where i = 1. Mark $\mathcal{M}_i$ as *used*
- **Step 3.** For each cluster $C_{curr,m}$ in the current metric axis M_{curr}, intersect with all clusters $C_{j,k}$ in one of the non *used* metric axis $\mathcal{M}_j$, $j \in \{1 .. 5\}$. The clusters in the resulting set contain potential code clone fragments under the criterion $\mathcal{M}_{curr}$ **and** $\mathcal{M}_j$, and form a composite metric axis $\mathcal{M}_{curr \odot j}$. Mark $\mathcal{M}_j$ as *used* and set the current axis $\mathcal{M}_{curr} = \mathcal{M}_{curr \odot j}$.
- **Step 4.** If all metric axes have been considered then stop; else go to *Step 3.*

The comparison granularity is at the statement level and for statements that are of length of more than n lines long, where n is a parameters provided by the user.

An interesting point of discussion is the threshold selection for each metric axis. The reason this issue is important is that each software system has unique metric characteristics and the distribution of values varies from system to system. Within this framework we have experimented with the following threshold selection options:

(i) fixed threshold values for every metric dimension

(ii) normalized threshold values. Normalized threshold distances are based on the premise that each metric dimension has its own characteristics and value ranges so that an acceptable threshold that may be used to differentiate one entity from another in a particular dimension is given by:

$$\frac{MaxValue - MinValue}{weight \cdot AverageValue} \qquad (4.1.1)$$

where *Min-value* and *Max-value* are the highest value and the lowest value seen in this metric dimension for the system under analysis respectively and *weight* is a adjusting parameter.

Metric Value	*Frequency*	Difference from Previous Value
0.0	108	0.0
1.0	94	1.0
2.0	104	1.0
3.0	68	1.0
....	....	....
33	1	1.0
35	1	2.0
....	....	....
58	1	5.0
67	1	9.0

TABLE 4.1. Step Distance table for S-Complexity taken from the Tcsh system

(iii) step distances. Step distance thresholding is based on the assumption that each metric dimension has different distribution of values for a subject system. For example the *S-Complexity* metric dimension has a minimum value-step of *1* while *Information Flow* and *Function Point* may exhibit a more complex pattern. Under this category we have experimented by using as an increase in threshold the next minimal difference between any two distances for all the code fragments compared. In Table.4.1 an example of data used for this type of threshold selection applied to the *s-complexity metric* dimension and for values obtained from the `tcsh` program is shown. In this example, the first minimal difference is 1.0, the next is 2.0, the third is 5.0 and so on.

1.2. Partition Clustering Clone Detection. The pattern matching engine uses the computed Euclidean distance and a clustering threshold value that is used as the clustering criterion. Euclidean distance is the most common distance discussed in the literature[**Hartigan75**]. Other distances include *Disguised Euclidean Distances*, the *Pearson distance* [**Pearson26**], the *Catell distance*[**Catell49**], the *Manhattan distance* and the *Mahalanobis* distance [**Mahala36**]. Each of these distances aims on addressing the complications that may arise because the variables lie on different scales or are of different types (i.e. election rates and personal income). In the framework of the software metrics we used, the variables after normalization, are in similar scales and of compatible types, and thus the application of the Euclidean distance yields the fastest and simplest solution.

In the literature [**Spath80**], [**Hartigan75**], [**Everitt74**], a number of clustering algorithms have been proposed (e.g. the *Leader algorithm*, the *Sorting algorithm*, the *K-Means* algorithm, as well a number of heuristic algorithms) [**Anderberg73**].

We have chosen a variation of the *Leader algorithm* suggested by [**Lu78**] based on the nearest-neighbor rule. The reason we used this algorithm is it is very fast, requiring only one pass through the data. The negative aspect of the algorithm is that the partition is not invariant under the reordering of the cases. This negative aspect can be eliminated if the clustering threshold distance is set to zero. Then, the partition will be be invariant under the reordering of cases as all elements of distance zero will eventually be contained in the same cluster.

Here we discuss a variation of this algorithm as is described in [**Jain88**]. Our variation marks the entries of the *Similarity Matrix* that have already been assigned to clusters. In this way search for the best distance between a pattern and a cluster is faster as the elements of the *Similarity Matrix* that have to be searched are fewer.

Let $\mathcal{P} = \{X_1, X_2, ...X_n\}$ be a set of patterns to be partitioned into K clusters.

- **Step 1.** Set i $\leftarrow$ 1 and k $\leftarrow$ 1. Assign pattern X_1 to cluster C_1
- **Step 2.** Set i $\leftarrow$ i+1. Find nearest neighbour of X_i aiming the patterns already assigned to clusters. Let d_m denote the distance from X_i to its nearest neighbour. Suppose that the nearest neighbour is in cluster m.
- **Step 3.** If $d_m \preceq t$, then assign X_i to C_m. Otherwise set k $\leftarrow$ k+1 and assign X_i to a new cluster C_k.
- **Step 4.** Delete all distance pairs between X_i and all patterns already assigned to clusters.
- **Step 5.** If every pattern has been assigned to a cluster, stop. Else, go to Step 2.

This partition clustering similarity analysis has been applied to a several medium-sized production C programs. Experimental results obtained by applying this technique are shown in Chapter.6.

2. Dynamic Programming Based Similarity Analysis

In addition to the direct metric comparison techniques, we propose dynamic programming techniques to calculate the best alignment between two code fragments based on *insertion*, *deletion* and *substitution* operations. Rather than working directly with textual representations, source code statements, are abstracted into feature sets that classify the given statement. Dynamic Programming and Tree matching has been proposed in [**Aho89**] for code generation. In [**Aho89**] rewrite rules are used to map tree-structures to assembly instructions. Dynamic Programming plays an importnt role on matching efficiently the "heads" of the rewrite rules to the tree-structures and thus apply the required transformation. However, this is different from our matching objectives. In the context of code cloning detection two ASTs that correspond to cloned code fragments may be quite different in structure, making thus the use of tree matching techniques very difficult and inefficient.

Within this framework, we use features to classify a statement and we use these features in the matching process. The statement features used in this Dynamic Programming approach are:

- Uses of variables, definitions of variables, numerical literals, and strings
- Uses and definitions of data types
- The five metrics as discussed previously

Once program features have been computed (at parse time), similarity between two statements takes the form of computed *coefficients* between the profiles of these statements. In [**Maarek91**] a number of such coefficients is discussed. These include *Dice's* coefficient, *Jaccard's* coefficient, and *Salton's* coefficient. These constitute standard coefficients in the literature, and within this framework we have experimented with *Dice's* coefficient, and *Jaccard's* coefficient. Both gave similar results.

Within this framework, Dynamic programming (DP) techniques detect the best alignment between two code fragments based on *insertion*, *deletion* and *substitution* operations. Two statements match if they *update* and *use* the same variables, strings, and numerical literals. Variations in these features at the simple statement [1] level provide a dissimilarity value used to calculate a global dissimilarity measure of more complex and composite constructs such as composite statements, `begin-end` blocks and, functions. The comparison function used to calculate dissimilarity measures is discussed in detail in the following Section. Heuristics have been incorporated in the matching process to facilitate variations that may have occurred in cut and paste operations. In particular, the following heuristics are currently used:

- Adjustments between variable names by considering lexicographical distances (i.e. maximum common subsequences in identifier names)
- Filtering out short and trivial variable names such as `i` and `j` which are typically used for temporary storage of intermediate values, and as loop index values. The user may provide the minimum length of a variable to be considered in the matching process as a parameter. Our experiments have been conducted by setting this minimum threshold to three characters long.

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 [**Konto94**], [**Konto95**]. Code fragments are selected for Dynamic Programming comparison by preselecting potential clone candidates using the direct metric comparison analysis. Within this framework only the source code statements that have a dissimilarity measure less than a given threshold are considered

[1] A simple statement is a source code statement that is not composed of other statements

for DP comparison. This preselection reduces the comparison space and increases efficiency as DP matching is more computationally expensive that the Metrics-based approach.

2.1. Similarity Distance Calculation. The distance between the two code fragments is given as a summation of comparison values as well as of insertion and deletion costs corresponding to insertions and deletions that have to be applied in order to achieve the best alignment between these two code fragments.

A program feature vector is used for the comparison of two statements. The features are stored as attribute values in a frame-based structure representing expressions and statements in the AST. We propose the following Dynamic Programming function with signature:

$$D : \textit{Feature\_Vector} \text{ x } \textit{Feature\_Vector} \rightarrow \textit{Real}$$

for computing the cumulative similarity measure $\mathcal{D}$ between two code fragments $\mathcal{P}$, $\mathcal{M}$. Specifically,

$$D(\mathcal{E}(1,p,\mathcal{P}),\mathcal{E}(1,j,\mathcal{M})) = Min \begin{cases} \Delta(p,,j-1,\mathcal{P},\mathcal{M})+ \\ D(\mathcal{E}(1,p,\mathcal{P}),\mathcal{E}(1,j-1,\mathcal{M})) \\ \\ I(p-1,j,\mathcal{P},\mathcal{M})+ \\ D(\mathcal{E}(1,p-1,\mathcal{P}),\mathcal{E}(1,j,\mathcal{M})) \\ \\ C(p-1,j-1,\mathcal{P},\mathcal{M})+ \\ D(\mathcal{E}(1,p-1,\mathcal{P}),\mathcal{E}(1,j-1,\mathcal{M})) \end{cases} \tag{2.1.1}$$

where,

- $\mathcal{M}$ is the model code fragment
- $\mathcal{P}$ is the input code fragment to be compared with the model $\mathcal{M}$
- $\mathcal{E}(i,j,\mathcal{Q})$ is a program feature vector from position i to position j in code fragment $\mathcal{Q}$
- $D(\mathcal{V}_x\ ,\ \mathcal{V}_y)$ is the the distance between two feature vectors $\mathcal{V}_x$, $\mathcal{V}_y$,
- $\Delta(i,j,\mathcal{P},\mathcal{M})$ is the cost of deleting the j*th* statement of $\mathcal{M}$, at position i of the fragment $\mathcal{P}$
- $I(i,j,\mathcal{P},\mathcal{M})$ the cost of inserting the i*th* statement of $\mathcal{P}$ at position j of the model $\mathcal{M}$ and
- $C(i,j,\mathcal{P},\mathcal{M})$ is the cost of comparing the i*th* statement of the code fragment $\mathcal{P}$ with the j*th* fragment of the model $\mathcal{M}$. The comparison cost is calculated by comparing the corresponding feature vectors. Currently, we compare ratios of variables updated, used per statement, data types used or set, and comparisons based on metric values

Note that *insertion*, and *deletion* costs are used by the Dynamic Programming algorithm to calculate the best fit between two code fragments. An intuitive interpretation of the best fit using *insertions* and *deletions* is "if we insert statement i of the input at position j of the model then the model and the input have the smallest feature vector difference"

The quality and the accuracy of the comparison cost is based on the program features selected and the formula used to compare these features. For simplicity in the implementation we have attached constant real values as insertion and deletion costs, to reflect the tolerance of the user towards partial matching. The meaning of the insertion and the deletion costs is discussed in the following paragraphs.

The comparison cost function $C(i, j, \mathcal{M}, \mathcal{P})$ is the key factor in producing the final distance result when DP-based matching is used. There are many program features that can be considered to characterize a code fragment (indentation, keywords, metrics, uses and definitions of variables). Within the experimentation of this approach we used the following three different categories of features

(i) updates and uses of variables as well as literal values within a statement (i.e. a message in a *printf* statement):
   (a) $Feature_{1_1} : Statement \rightarrow \{String\}$ denotes the set of variable names used within a statement,
   (b) $Feature_{1_2} : Statement \rightarrow \{String\}$ denotes the set of variable names updated[2] within a statement
   (c) $Feature_{1_3} : Statement \rightarrow \{String\}$ denotes the set of literal values (e.g. numbers, strings) within a statement (e.g. a *printf* statement).

(ii) definitions and uses of data types :
   (a) $Feature_{2_1} : Statement \rightarrow String$ denotes the set of data type names used within a statement,
   (b) $Feature_{2_2} : Statement \rightarrow String$ denotes the set of data type names updated within a statement

The comparison cost of the ith statement in the input $\mathcal{P}$ and the jth statement of the model $\mathcal{M}$ for the first two categories is calculated as :

$$C(\mathcal{P}_i, \mathcal{M}_j) = \frac{1}{v} \cdot \sum_{m=1}^{v} \frac{card(InputFeature_m(\mathcal{P}_i) \cap ModelFeature_m(\mathcal{M}_j))}{card(InputFeature_m(\mathcal{P}_i) \cup ModelFeature_m\mathcal{M}_j))} \qquad (2.1.2)$$

[2]Also referred to in the literature as: defined, or set or, stored

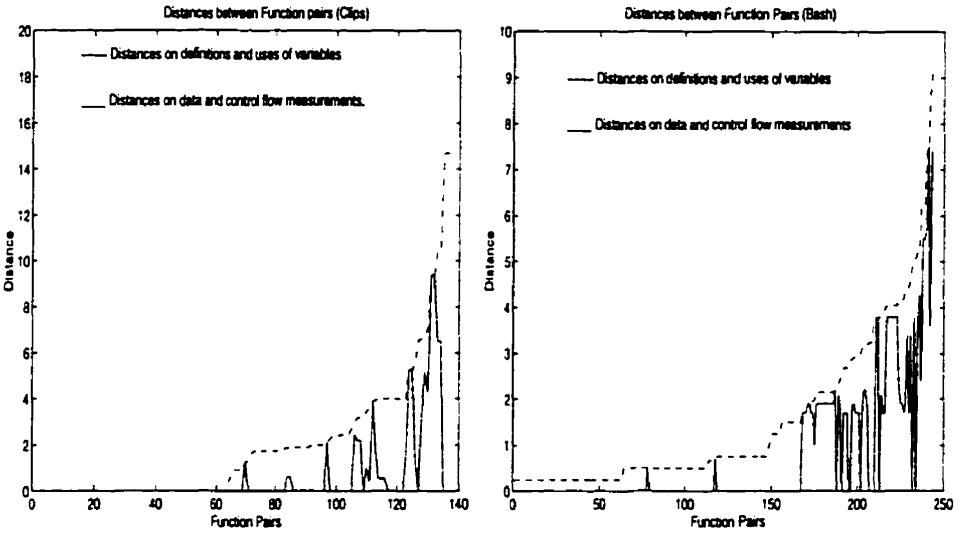


FIGURE 4.1. Distances between function pairs of possible function clones for the Clips and Bash programs using DP-based matching. The dashed line represents measurements obtained using the *set-uses* criterion. The solid line represents measurements obtained by the *metrics* criterion. The values in the X - axis represent the n*th* function pair that has been identified as containing potential clones (i.e. the two functions have zero distance) using the metric comparison similarity analysis.

where v is the size of the feature vector, or in other words how many features are used,

(iii) five metric values which are calculated compositionally from the statement level to function level as discussed in Section.2 :

The comparison cost of the i*th* statement in the input $\mathcal{P}$ and the j*th* statement of the model $\mathcal{M}$ when the five metrics are used is calculated as :

$$C(\mathcal{P}_i, \mathcal{M}_j) = \sqrt{\sum_{k=1}^{5} (M_k(\mathcal{P}_i) - M_k(\mathcal{M}_j))^2} \qquad (2.1.3)$$

Within this framework new metrics and features can be used to make the comparison process more sensitive and accurate.

Moreover, the following points on *insertion* and *deletion* costs need to be discussed.

- The *insertion* and *deletion* costs reflect the tolerance of the user towards partial matching (i.e. how much noise in terms of *insertions* and *deletions* is allowed before the matcher fails). Higher *insertion* and *deletion* costs indicate smaller tolerance, especially if cut-off thresholds are used (i.e. terminate matching if a certain threshold is exceeded), while smaller values indicate higher tolerance.

- The values for *insertion* and *deletion* should be higher than the threshold value by which two statements can be considered *"similar"*, otherwise an *insertion* or a *deletion* could be chosen instead of a *match*.
- A lower *insertion* cost than the corresponding *deletion* cost indicates the preference of the user to accept a code fragment $\mathcal{P}$ that is written by inserting new statements to the model $\mathcal{M}$. The opposite holds when the *deletion* cost is lower than the corresponding *insertion* cost. A lower *deletion* cost indicates the preference of the user to accept a code fragment $\mathcal{P}$ that is written by deleting statements from the model $\mathcal{M}$. *Insertion* and *deletion* costs are constant values throughout the comparison process and can be set empirically.

When different comparison criteria are used different distances are obtained. In Fig.4.1 distances calculated using Dynamic Programming are shown. For example in Fig.4.1, the distances obtained using the Metrics and the Set-Uses criterion for the Clips program are illustrated.

The dashed line shows distance results when *updates* and *uses* of variables are used as features in the dynamic programming approach, while the solid line shows the distance results obtained when the five metrics are used as features.

Table 4.2 summarizes statistical data regarding false alarms when Dynamic Programming comparison between functions in Clips was applied. The column labeled *Distance Range* gives the value range of distances between functions using the Dynamic Programming approach. The column labeled *False Alarms* contains the percentage of functions that are not clones but they have been identified as such. The column labeled *Partial Clones* contains the percentage of functions which correspond only to partial cut and paste operations. Finally, the column labeled as *Positive Clones* contains the percentage of function clearly identified as cut and paste operations.

As an example consider the following statements $\mathcal{M}$ and $\mathcal{P}$:

- $\mathcal{M}$:

```
ptr = head;
while(ptr != NULL && !found)
 { if(ptr->item == searchItem)
     found = 1
   else
     ptr = ptr->next;
 }
```

- $\mathcal{P}$

```
while(ptr != NULL && !found)
```

Distance Range	*False Alarms*	*Partial Clones*	*Positive Clones*
0.0	0.0 %	10.0%	90.0%
0.01 - 0.99	6.0 %	16.0 %	78.0%
1.0 - 1.49	8.0%	3.0 %	89.0%
1.5 - 1.99	30.0%	37.0 %	33.0%
2.0 - 2.99	36.0%	32.0 %	32.0%
3.0 - 3.99	56.0%	13.0 %	31.0%
4.0 - 5.99	82.0%	10.0 %	8.0%
6.0 - 15.0	100.0%	0.0 %	0.0%

TABLE 4.2. False alarms for the Clips program using DP matching and the Set-Uses criterion.

```
{ if(ptr->item == searchItem)
    { printf("ELEMENT FOUND : %s\n", searchItem);
      found = 1;
    }
  else
    ptr = ptr->next;
}
```

The Dynamic Programming matching based on definitions and uses of variables is illustrated in Fig. 4.2.

In the first grid the two code fragments are initially considered. At position (0, 0) of the first grid a deletion is considered as it gives the best cumulative distance to this point (assuming there will be a match at position (0, 1). The comparison of the two composite *while* statements in the first grid at position (0, 1), initiates a nested match (second grid). In the second grid the comparison of the composite `if-then-else` statements at position (1, 1) initiates a new nested match. In the third grid, the comparison of the composite `then-part` of the `if-then-else` statements initiates the final fourth nested match. Finally, in the fourth grid at position (0, 0), an insertion has been detected, as it gives the best cumulative distance to this point (assuming a potential match at position (1, 0)).

When a nested match process finishes it passes its result back to the position from which it was originally invoked and the matching continues from this point on.

The DP technique has been successfully applied to detect code cloning and facilitate partial matching. The Section below, summarizes another use of code cloning detection when combined with generic data flow analysis, and namely, system partitioning.

2.2. System Partitioning. A large software system is very difficult to analyse as a whole. One solution is to decompose the system according to a number of criteria. System Partitioning is,

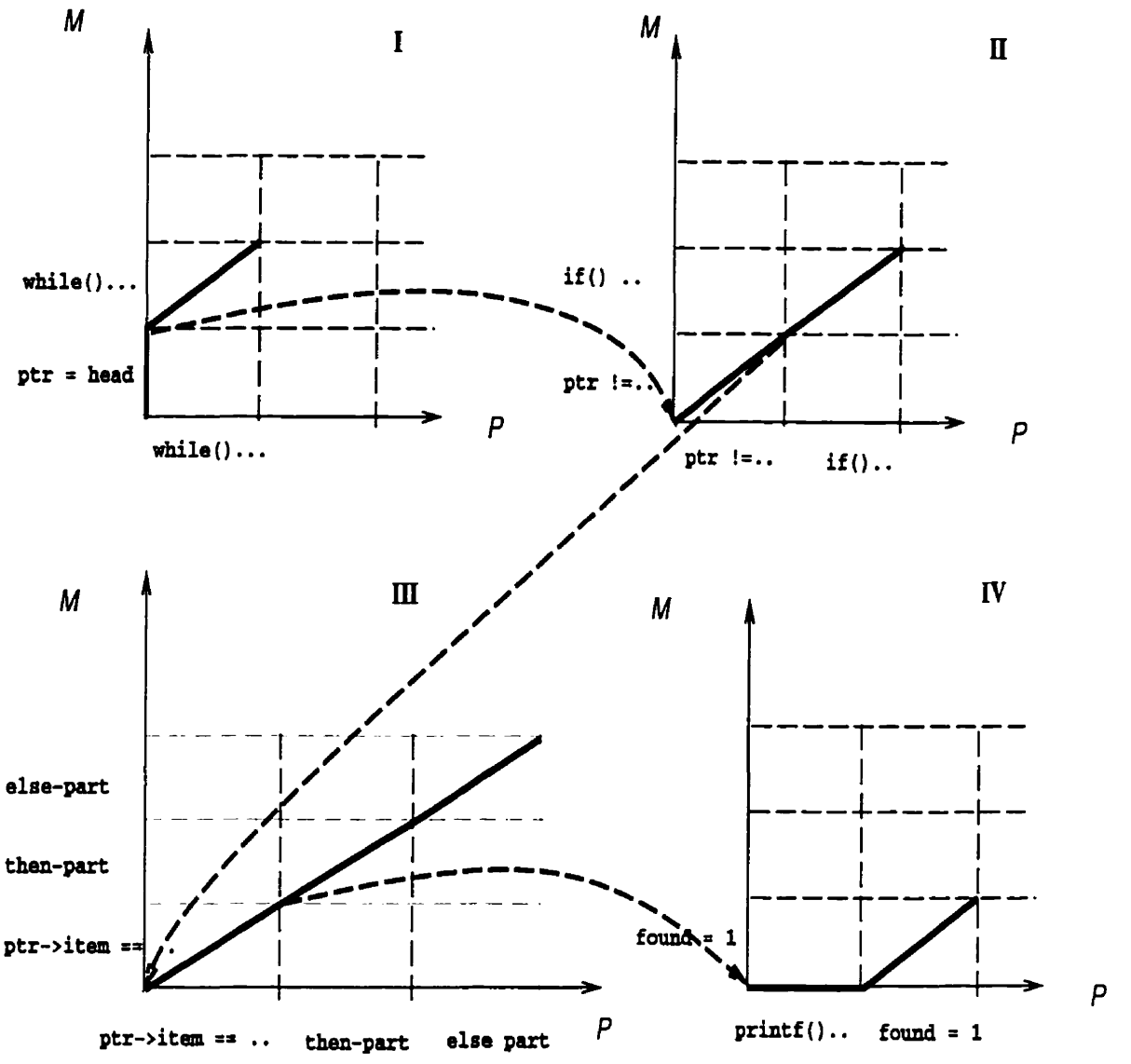


FIGURE 4.2. The matching process between two code fragments. Insertions are represented as horizontal lines, deletions as vertical lines and, matches as diagonal lines.

on its own, a large area of research and it is not the focus of this thesis but, nevertheless, we would like to devote a small section to ideas that we found to be practical when analysing a large system [**Buss94**].

System Partitioning is a necessary step when analysing a large software system, that due to space or time limitations, can not be treated as a whole. To perform System Partitioning key points that have to be addressed include:

(i) The Partitioning criteria (i.e. what is a successful partitioning)

(ii) The Partitioning features (i.e. what are the available program features to perform Partitioning on)

(iii) The initial state from which the Partitioning process will start

Partitioning and grouping criteria may include:

- Maximise cluster size and minimise inter-cluster data and control flow (this is an optimisation problem)
- All instances of a particular data type (i.e. a date field) are included in a partition
- Functionality (e.g. a part of a large application that implements a particular task)
- Customer imposed criteria

Partitioning and grouping features may include:

- Structural Similarity
- Code affected using impact analysis and starting from a set of initial requirements (i.e. a set of variables)
- Data types fetched or stored
- Access to external sources (i.e. Data Files)
- Customer imposed features

The features we experimented with are :

- Data Bindings Analysis [3]
- Common Resources Analysis

Data Bindings analysis focuses on the identification of triplets $< C_1, C_2, V >$ where C_1, C_2 are sets of functions and V a set of variables such that all functions in C_1 define all variables in V and all functions in C_2 use all variables in V. This type of analysis allows for the identification of modules or subsystems whose components have high coupling. When definitions and uses of variables are computed, basic aliasing (i.e. parameters passed by reference), is considered as well.

Common References analysis focuses on the identification of pairs $< C_1, V >$ where C_1 is a set of functions and V a set of variables such that all functions in C_1 define or use all variables in V. This type of analysis considers only the variable name and type, and no aliasing or scoping information is taken into account. The assumption is that if two functions have variables in common with the same name and type, most probably they refer to the same concept. This type of analysis reveals modules or subsystems that are related by some concept and data type. This type of analysis has been used by Rigi to visualise segmentations of various systems [**Buss94**]. This type of segmentation reveals modules at the architectural level. In Fig.4.3 a segmentation of the CLIPS system based on

[3] Data Bindings analysis has originally proposed by R. Selby and V. Basili [**Selby90**] as a method for identifying error prone structures (i.e. structures that have a very high data flow dependencies

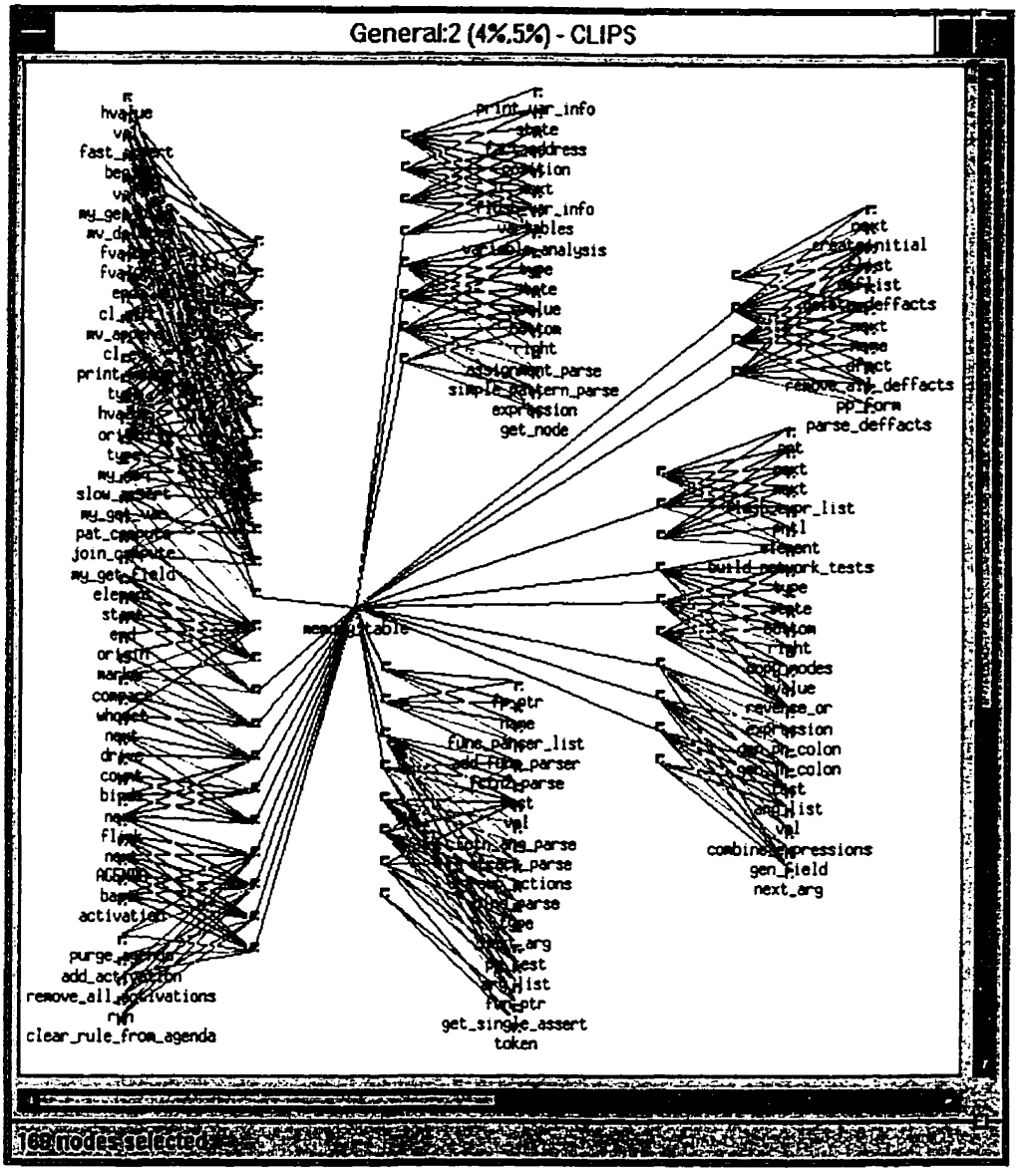


FIGURE 4.3. Segmentation of the Clips System using Clustering on Data Bindings, Common References, and Code Cloning

data bindings and common references analysis is illustrated. In the example ilustrated, anumber of different modules have been identified. These include the *fact management module* which contains functions *remove_deffacts*, *parse_deffacts*, *remove_all_deffacts*, the *rule management module* which contains functions *clear_rule_from_agenda*, *remove_all_activations*, *purge_agenda*, *add_activation*, and the *variable bindings module* which contains functions *print_var_info*, *fact_address*, *position*, and *variable_analysis*. This segmentation was displayed in the Rigi environment [**Muller91**] (see Section.1).

CHAPTER 5

Concept To Code Matching

The *concept assignment* [**Biggerstaff94**], [**Biggerstaff89**] problem consists of assigning concepts described in a concept language to program fragments. Concept assignment can also be seen as a matching problem. In our approach, concepts are represented as *abstract-descriptions* using a *concept language* called *Abstract Concept Language* (ACL). It is assumed that a concept description may match a number of different implementations represented by program segments in source code called *code fragments*. The similarity between a description and a code fragment is measured by their matching probability. In this framework, an *abstract-description* is parsed and a corresponding AST T_a is created. Similarly, source code is represented by an annotated AST T_c. Both T_a and T_c are transformed into a sequence of abstract and source code statements, respectively, using transformation rules. These rules transform a part of the AST that represents a source code statement to a sequence of entities that the statement is composed of. For example, an `IF-THEN-ELSE` statement is transformed to a sequence [*Condition, Then-Part, Else-Part*]. The objective is to reduce the complexity of the matching algorithm as T_a and T_c may have very complex structures. In this way structural details of the ASTs have been abstracted and represented as sequences of entities.

An overview of the matching process is illustrated in Fig.5.1. The abstract pattern written in ACL is parsed and transformed into a Markov Model. Similarly, source code is represented as a sequence of statements. The Viterbi algorithm is used to find the best fit between the model and the sequence of input statements using feature vectors that represent data flow, control flow, and informal information properties of the code.

Problems of matching concepts to code that have been considered are:

- The choice of the conceptual language,
- The measure of similarity,
- The selection of a fragment in the code to be compared with the conceptual representation.

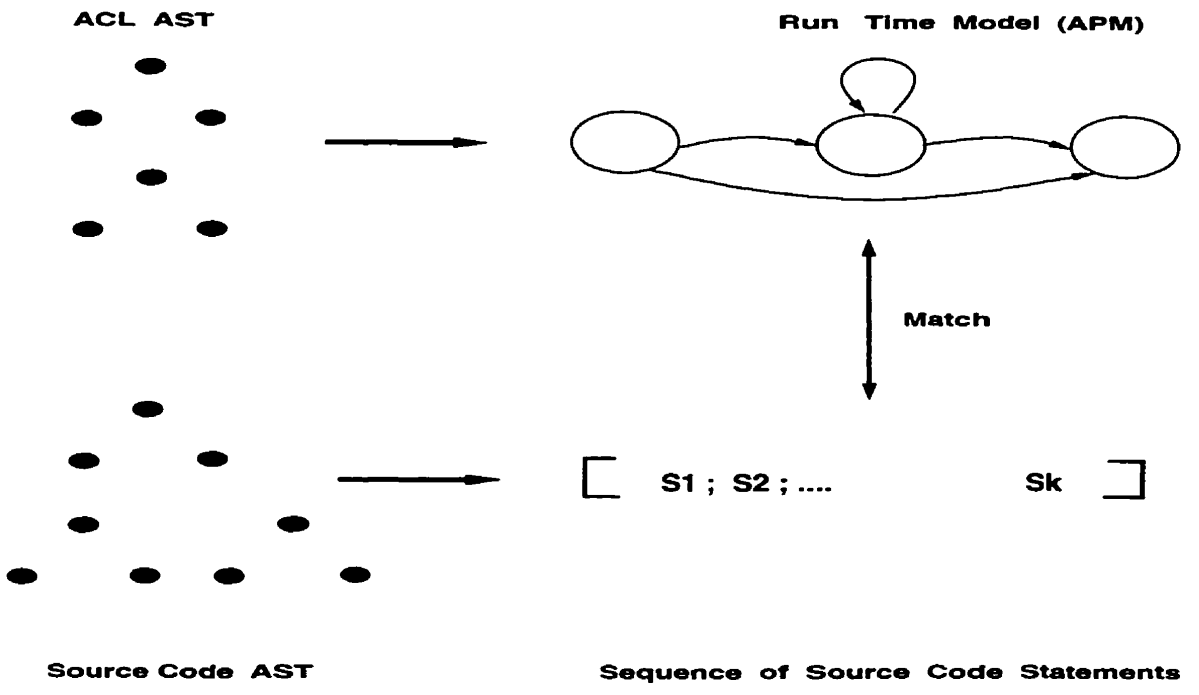


FIGURE 5.1. Overview of the Markov-Based Code Matching Process

These problems are addressed in the following sections.

1. Language for Abstract Representation

A number of research teams have investigated the problem of code and plan localization. Current successful approaches include the use of graph grammars [**Wills92**], [**Rich90**], query pattern languages [**Paul94**], [**Muller93**], [**Biggerstaff94**], sets of constraints between components to be retrieved [**Ning94**], and summary relations between modules and data [**Canfora92**].

The proposed approach focuses on facilitating partial matching, a situation that is frequent in practice and has yet been addressed in a framework of uncertainty reasoning. We propose in this context a novel approach: a stochastic pattern matcher that allows for partial and approximate matching within the context of *Plan Recognition*. A concept language [1] represents, in an abstract way, sequences of design concepts corresponding to a "design pattern".

We view ACL patterns as a program representation tool which:

(i) decomposes the program representation into relationships (stores, fetches, used-by, calls, called-by, keywords, metrics),

(ii) allows for structural and data type abstraction,

(iii) deals with syntactic and implementation variations and,

[1] Please refer to Appendix A, Appendix B, for a complete description of the ACL grammar and domain model

(iv) allows for representing noncontiguous plans.

The concept language contains:

- *Abstract statements* $\mathcal{S}$ that may match (generate) one or more statement types in the source code language. The correspondence between an *abstract statement* and the source code statements it may generate is given at Table 5.1.

  ACL contains the following abstract statements:

  (i) Abstract Iterative Statements
    (a) Abstract While Statement
    (b) Abstract For Statement
    (c) Abstract Do Statement

  (ii) Abstract Conditional Statements
    (a) Abstract If Statement
    (b) Abstract Switch Statement

  (iii) Abstract Expression Statements
    (a) Abstract Function Calls
    (b) Abstract Assignments
      (i) Abstract Actual Assignment
      (ii) Abstract Post/PreIncrementation
      (iii) Abstract Post/Decrementation

  (iv) Abstract Return Statement

  (v) Abstract GoTo Statement

  (vi) Abstract Continue Statement

  (vii) Abstract Break Statement

  (viii) Abstract Labeled Statement

  (ix) Abstract *Statement**

  (x) Abstract *Statement*$^+$

  (xi) Abstract Any-Statement.

  (xii) Inline-Plan-Statement.

- *Abstract Expressions* $\mathcal{E}$ that correspond to source code expression. The correspondence between an *abstract expression* and the source code expression that it may generate is given in Table 5.2. *ACL* contains the following abstract expressions:

  (i) Abstract Equality

  (ii) Abstract Inequality

ACL Statement	Generated Code Statement
Abstract Iterative Statement	*While Statement* *For Statement* *Do Statement*
Abstract While Statement	*While Statement*
Abstract For Statement	*For Statement*
Abstract Do Statement	*Do Statement*
Abstract Conditional Statement	*If Statement* *Switch Statement*
Abstract If Statement	*If Statement*
Abstract Switch Statement	*Switch Statement*
Abstract Return Statement	*Return Statement*
Abstract GoTo Statement	*GoTo Statement*
Abstract Continue Statement	*Continue Statement*
Abstract Break Statement	*Break Statement*
Abstract Labeled Statement	*Labeled Statement*
AbstractStatement*	**Zero or more sequential source code statements**
AbstractStatement$^+$	**One or more sequential source code statements**
Abstract Any-Statement	**One occurrence of a source code statement**

TABLE 5.1. Generation (Allowable Matching) of source code statements from ACL statements

(iii) Abstract Logical-And

(iv) Abstract Logical-Or

(v) Abstract Logical-Not

(vi) Abstract Function-Call

(vii) Abstract Identifier

- Abstract-Named-Identifier

ACL Expression	Generated Code Expression
Abstract Function Call	*Function Call*
Abstract Equality	*Equality (==)*
Abstract Inequality	*Inequality (!=)*
Abstract Logical And	*Logical And (&&)*
Abstract Logical Or	*Logical Or (||)*
Abstract Logical Not	*Logical Not (!)*
Abstract Any-Expression	*Any Source Code Expression*

TABLE 5.2. Generation (Allowable Matching) of source code expressions from ACL expressions

  - Abstract-Variable-Identifier

- *Abstract Feature Descriptors* $\mathcal{F}$ that contain the feature vector data used for matching purposes. Currently the features that characterize an *abstract statement* and an *abstract expression* are:

  (i) *Uses of variables* : variables that are used in a statement or expression,

  (ii) *Definitions of variables*: variables that are defined in a statement or expression,

  (iii) *Keywords* : strings, numbers, characters that may used in the text of a code statement,

  (iv) *Metrics* : a vector of five different complexity, data and control flow metrics.

- *Abstract Identifiers* $\mathcal{X}$

  - *Abstract Variable Identifiers* are used as place-holders for feature vector values, when no actual values for the feature vector are provided. An example is when we are looking for a *Traversal of a linked list* plan but we do not know the names of the pointer variables that exist in the code.
  - *Abstract Named Identifiers* are more restrictive in the sense that the matching identifier in the source code has to have a similar name. By similar name we mean the lexicographical distance between the two names is below a certain threshold that can be adjusted by the user. Zero lexicographical distance means that the two identifiers have the same name.

Default ACL Type	Generated Code Type
Struct	*struct*
Array	*array ([])*
Numeral	*float* or *int*
Character	*char*
Any-type	*any C type*
named	*the particular named type*

TABLE 5.3. Generation (Allowable Matching) of source code data types from ACL data types

- *Abstract Data Types* $\mathcal{T}$ An Abstract Data Type t, associated with an Abstract Identifier can generate (match) any actual type in the source code provided that they belong to the same data type category. For example a *Struct* type abstract variable can be matched with a *struct* source code variable in *C* or a *Record* in *Pascal*. The power of the approach lies in the fact that Abstract Data Types are essentially object classes and the user can specify his or her own hierarchies. For example a new ADT *List* may be defined as a superclass of the *Array* and *Struct* ADT and thus allow for matching with an *array* or *struct* actual source code data type.

  The system supports by default the following abstract types:

  (i) Structure : Representing *struct* types,

  (ii) Array : Representing *array* types,

  (iii) Numeral : Representing *int*, and *float* types,

  (iv) Character : Representing *char* types,

  (v) Any-type : Representing any source code type types,

  (vi) Named : matching the actual data type name in the source code,

  and the following access methods:

  (i) Pointer : Representing a pointer identifier,

  (ii) Reference : Representing a simple identifier reference.

  The correspondence between an *abstract data type* and the source code type that it may generate is given in Table 5.3.

- *Operators* $\mathcal{O}$

  Operators are used to compose abstract statements from simpler ones:

(i) Sequencing (;) : To indicate that one statement follows another

(ii) Choice ($\oplus$) : To indicate choice (one or the other abstract statement will be used in the matching process)

(iii) Inter Leaving ($\|$) : to indicate that two statements can be interleaved during the matching process

- *Macros* $\mathcal{M}$

  Macros are proposed to facilitate hierarchical plan recognition [**Hartman92**], [**Chiko90**]. Macros are entities that refer to plans that are in-lined in the pattern when the AST for the pattern is created. For example if a plan has been recognized is stored in the plan base, then special preprocessor statements are used to include this plan to compose more complex patterns. Included plans are incorporated in a pattern's AST at parse time in a way similar to C++ *in-line* functions.

  Special macro definition statements in the Abstract Language are used to include the necessary macros.

  We consider two types of Macro related statements in ACL:

  (i) Macro declarations: These are special statements in *ACL* that specify the name of the plan to be included and the file where it is defined.

  As an example consider the ACL statement

  `include` *plan1.acl* *traversal-linked-list*

  that imports the plan *traversal-linked-list* which is defined in file *plan1.acl.*

  (ii) Macro uses : These are statements that direct the parser to *in-line* the particular plan and include its AST in the original pattern's AST. As an example, consider the following *Abstract Macro use*:

  `plan:` *traversal-linked-list*

  that is used to include an instance of the *traversal-linked-list* plan at a particular point of the pattern. In a pattern more than one occurrence of an included plan may appear.

A typical example of a design concept in our concept language is given below. This pattern expresses an iterative statement (e.g. *while* , *for*, *do* loop that has in its condition an inequality expression that uses the identifier `?x` that is a pointer to an abstract type `struct` and the conditional expression contains the keyword "NULL". The body of the `Iterative-Statement` contains a sequence of one or more statements (`+-Statement`) and an `Assignment-Statement` that uses at least identifier `?x`, defines identifier `?x`. However, this example pattern limits the scope of an *Iterative-Statement* to be matched only with a *While-Statement* or a *For-Statement.* Moreover, this patterns

requires that the probability of a *While-Statement* appearing is 0.75, and that the probability of a *For-Statement* appearing in the source code is 0.25[2]. If no probability preferences are given by the user the system assumes a uniform distribution where the a *while* , *for*, *do* loop have equal occurrence probabilities (i.e. 0.33).

The bindings that may occur for a successful match between this ACL pattern and the code fragment given below is {?x/field}. Note that the binding {?x/pos} that is activated at the condition of the *while* statement when matched with the condition of the ACL *Iterative-Statement* is eliminated when the *field = field→nextValue* statement is matched with the ACL *Assignment-Statement* in the example pattern.

```
probability : ["While-Statement, 0.75, "For-Statement", 0.25]
Iterative-Statement(Inequality-Expression
                            uses : [ ?x : *struct],
                            keywords : [ "NULL" ])
        {
          +-Statement;
          Assignment-Statement
               uses : [?x],
               defines : [?x],
               keywords : [ "next" ]
        }
```

A code fragment that matches the pattern is:

```
while (field != NULL && pos == 1)
  {
   if (!strcmp(obj,origObj) ||
       (!strcmp(field->AvalueType,"member") &&
        notInOrig ) )
     if (strcmp(field->Avalue,"method") != 0)
       INSERT_THE_FACT(o->ATTLIST[num].Aname,origObj,
                    field->Avalue);
  field = field->nextValue;
  }
```

2. Abstract Language Semantics

In this section we discuss in more detail the constructs and the semantics of the abstract pattern language (ACL). As discussed above, the Abstract Concept Language consists of :

[2]Please refer to Section.5.4 for a complete description of static probability usage.

(i) Abstract Statements $\mathcal{S}$

(ii) Abstract Expressions $\mathcal{E}$

(iii) Feature Descriptions $\mathcal{F}$

(iv) Abstract Identifiers $\mathcal{X}$

(v) Operators $\mathcal{O}$

(vi) Macros $\mathcal{M}$

We define the semantics of *ACL* entities in terms of a *semantic function* S_d defined as follows:

$$S_d : \mathcal{A} \rightarrow A$$

where $\mathcal{A} = \mathcal{S} \cup \mathcal{E}$ is the domain of ACL abstract statements $\mathcal{S}$ and ACL abstract expressions $\mathcal{E}$. Similarly, $A = S \cup E$ is the domain of source code statements S and source code expressions E.

Let $AbsStat_{\mathcal{V}_x}(AbsExp_{\mathcal{V}_y})$ be the abstract statement *AbsStat* that is represented by the feature vector $\mathcal{V}_x$, and contains the abstract expression *AbsExp*. Similarly let $AbsExp_{\mathcal{V}_y}$ be the the abstract expression *AbsExp* that is represented by the feature vector $\mathcal{V}_y$. The same notation holds for $SourceStat_{\mathcal{V}_f}$ and $SourceExp_{\mathcal{V}_g}$ for the source code statements and expressions respectively.

- *Abstract Statements* The semantics of an abstract statement $AbsStat_{\mathcal{V}_x}$ that contains the abstract expression $AbsExp_{\mathcal{V}_y}$, are given as:

$$S_d(AbsStat_{\mathcal{V}_x}(AbsExp_{\mathcal{V}_y})) = SourceStat_{S_d(\mathcal{V}_x)}(S_d(AbsExp_{\mathcal{V}_y}))$$

such that:

SourceStat $\in$ *Gen*(*AbsStat*) and *Gen*(*AbsStat*) is a mapping that denotes the possible source code statements that can be generated by *AbsStat*. This mapping is illustrated in Table.5.1.

- *Abstract Expressions*

$$S_d(AbsExp_{\mathcal{V}_y}) = SourceExp_{S_d(\mathcal{V}_y)}$$

such that:

SourceExp $\in$ *Gen*(*AbsExp*),

$AbsExp_{\mathcal{V}_y}$ denotes an abstract expression with features given by the feature vector $\mathcal{V}_y$ and $SourceExp_{S_d(\mathcal{V}_y)}$ denotes an actual source code expression that can be generated (matched) by $AbsExp_{\mathcal{V}_y}$.

As above, *Gen(AbsExp)* gives the possible source code expressions that can be generated by *AbsExp*. Table 5.2 illustrates the possible ways an abstract expression may generate a source code expression.

- *Feature Descriptors*

  The language supports the following features for every Abstract Statement *AbsStat* and Abstract Expression *AbsExp* :

  - Uses of variables (fetches)
  - Definitions of variables (stores [3])
  - Keywords (comments, approximate variable names)
  - Metrics (S-Complexity, D-Complexity, McCabe, Albrecht, Kafura)

  The semantics of each feature for an Abstract Statement or Expression P are given as :

  - $S_d(Uses(x_1, x_2, ...x_k)) = S_d(x_1)$ and $S_d(x_2)$ ... and $S_d(x_k)$

    where $x_1, ..x_k$ are abstract ACL Identifiers and $S_d(x_i) \in$ IDS_USED($S_d(P)$) (see Chapter.2).

  - $S_d(Defines(x_1, x_2, ...x_k)) = S_d(x_1)$ and $S_d(x_2)$ ... and $S_d(x_k)$

    where $x_1, ..x_k$ are abstract ACL Identifiers and $S_d(x_i) \in$ IDS_UPDATED($S_d(P)$) (please see Section.3.2).

  - $S_d(Keywords(str_1, str_2, ...str_k)) = S_d(str_1)$ and $S_d(str_2)$ ... and $S_d(str_k)$

    where $str_1, ..str_k$ are ACL string literals and $S_d(x_i) \in$ IDS_UPDATED($S_d(P)$) $\cup$ IDS_USED($S_d(P)$) $\cup$ LITERALS($S_d(P)$)[4] (please see Section.3.2).

  - $S_d(Metrics(m_1, m_2, ..m_5))$ = [S-COMPLEXITY($S_d(P)$), D-COMPLEXITY($S_d(P)$), McCABE($S_d(P)$), ALBRECHT($S_d(P)$), KAFURA($S_d(P)$)].

- *Abstract Identifiers*

  - *Named Identifiers*

    $S_d(q)$ = i, where q is an ACL *Named Identifier* and i is an identifier in the source code with the same name as q

  - *Variable Identifiers*

[3] Also referred to as *updates*

[4] LITERALS refers to the set of string values, character constants, and numerical constants that may occur in a source code statement. This set is computed in a compositional way on the AST nodes at the same way as the rest of the features are.

$S_d(?q)$ = i **and** the binding {?q/i} is added to the existing bindings of *?q*. Here, *?q* is an ACL *Variable Identifier* and *i* is an identifier in the source code.

- *Abstract Data Types*

  Abstract Data Types are always associated with an *Abstract Identifier*. The semantics of a *Typed Abstract Identifier* x of Abstract Data Type t are given from a mapping

  $$S_d(x : t) = S_d(x) : S_d(t)$$

  where $S_d(x)$ is an actual source code variable of type $S_d(t)$ that is compatible with data type t. Data type compatibility is described in Table 5.3. The user may enhance this default compatibility by adding new entries and ADT hierarchies as discussed previously.

- *Operators*

  The semantics for these operators are given as follows :

  - Sequencing

    $$S_d(AbsStat1_{\mathcal{V}_x}; AbsStat2_{\mathcal{V}_{x'}}) =$$

    $$S_d(AbsStat_{\mathcal{V}_x}); S_d(AbsStat2_{\mathcal{V}_{x \circ x'}})$$

    where :

    $x \circ x'$ means that any bindings that may have been generated for variables in $\mathcal{V}_x$ are applied to same occurrences of variables in $\mathcal{V}_{x'}$. For example if an *Abstract-Identifier* *?i* in $\mathcal{V}_x$ has been bound to source code variable i_1 then all occurrences of *?i* in $\mathcal{V}_{x'}$ will be bound to i_1 before the matching process for *AbsStat2* starts.

  - Choice[5]

    $$S_d(AbsStat1_{\mathcal{V}_x} \oplus AbsStat2_{\mathcal{V}_{x'}}) =$$

    $$S_d(AbsStat1_{\mathcal{V}_x}) \textbf{xor} S_d(AbsStat2_{\mathcal{V}_{x'}})^{6}$$

  - Interleaving[7]

    $$S_d(AbsStat1_{\mathcal{V}_x} | AbsStat2_{\mathcal{V}_{x'}}) =$$

[5]The *Choice* operator is defined as left associative
[6]**xor** denotes the standard "exclusive or" operation
[7]When more than two operands occur in an *Interleaving* expression all the permutations these operands can generate are considered.

$$S_d(AbsStat1_{\mathcal{V}_x}); S_d(AbsStat2_{\mathcal{V}_{x \circ x'}}) \oplus$$

$$S_d(AbsStat2_{\mathcal{V}_{x'}}); S_d(AbsStat1_{\mathcal{V}_{x' \circ x}})$$

- *Inline Macros*

  Let an in-lined Macro $M(param_1, param_2, param_k) = (A_{1_{\mathcal{V}_1}} .. op A_{n_{\mathcal{V}_n}})$ where $A_1, A_2, ...A_n$ are ACL statements and $param_1, param_2, ..param_k$ are ACL identifiers that exist in the feature vectors $\mathcal{V}_1, \mathcal{V}_2, ..\mathcal{V}_n$ of $A_1, A_2, ...A_n$ respectively. Similarly, *op* is any of the language operators (; || and, $\oplus$).

  Its semantics are given by

$$S_d(M(param_1, param_2, param_k)) = S_d(A_{1_{\mathcal{V}_1}} .. op A_{n_{\mathcal{V}_n}})$$

3. Concept-to-Code Distance Calculation

In this section we discuss the matching of an abstract pattern in *ACL* with source code.

In general the matching process contains the following steps :

(i) Source code $(S_1; ...S_k)$ is parsed and an AST T_c is created,

(ii) The ACL pattern $(A_1; ...A_n)$ is parsed and an AST T_a is created,

(iii) A transformation program generates from T_a a Markov Model called Abstract Pattern Model (APM),

(iv) A Static Model called SCM provides the legal entities of the source language. This Markov Model underlying finite-state automaton for the mapping between an *APM* state and an *SCM* model basically implements the Tables 5.1, 5.2 and, 5.3,

(v) Candidate source code sequences are selected according to a set of code delineation criteria discussed below,

(vi) The Viterbi [**Vite67**] algorithm is used to find the best fit between the Dynamic Model, resulting from the combination of APM and SCM, and a code sequence selected from the candidate list.

A Markov model is a source of symbols of an alphabet V characterized by states and transitions. A model can be in a state with a certain probability. From a state, a transition to another state can be taken with a given probability. A state is associated with the generation (recognition) of a symbol with a specific probability. The intuitive idea of using Markov models to drive the matching process is that an abstract pattern given in *ACL* may have many possible alternative ways to generate (match) a code fragment. A Markov model represents all these alternative options and assign to each of them a probability. Moreover, the Viterbi algorithm provides an efficient way to find the

path that maximizes the overall generation (matching) probability of a given string in V^* among all the possible alternatives.

Thus, a Markov Model has to be generated for a given ACL pattern. This APM model is then "augmented" by replacing each APM state with a corresponding SCM, as shown in Fig.5.2 where A_i and M_{i_j} are states and P_{ij} are transition probabilities.

Once an ACL pattern is parsed and the corresponding APM and SCM model created, the problem is to select candidate starting points for matching in a large software system. This is a delineation problem. The simple but most expensive solution is to consider every source code statement as a potential matching starting point and have a moving "window" of n many statements maintained at any given point. This is a very time and space consuming approach because of all the possible combinations it generates. Another approach is proposed in [**Fickas79**] and is based on "beacons" to hypothesize the existence of potential links between a plan (cliche) and its location in the source code, and statistical formalisms to guide *island-driven-parsing* [**Corazza90**]. The drawback of using "beacons" is that a plan decomposition must be assumed in advance using a "backward-chaining" strategy to hypothesize the next potential plan and the potential points in the code that this plan may be implemented. Similarly, the use of *island-driven-parsing* requires the existence of an accurate feature vector, a situation which is not always valid when forming an arbitrary ACL pattern. We propose that a simple but effective in the context of code delineation,[8], selection of a code fragment $S_1; S_2; ...S_k$ to be matched with an abstract description based on both of the following three criteria :

(i) the first source code statement S_1 matches with the first pattern statement A_1 and,

(ii) $S_2; S_3; ..; S_k$ belong to the inner most block $\mathcal{B}$, containing S_1

(iii) S_k is the last statement in $\mathcal{B}$.

The process starts by selecting for every `begin-end` block in a software system all potential starting points (i.e. source code statements), that match the first criterion above.

Matching according to the first criterion means that the types and the feature vectors of S_1 and A_1 give a high similarity probability [9] Once a candidate starting point or points have been selected for each `begin-end` block then, the sequences of statements starting from the selected points to the end of each block are returned (criterion 2, and 3). Note that this is a delineation process. The final code segmentation (i.e. the matched code) is produced by the matching process, and will contain the statement sequence $S_1; S_2; ...; S_i$, where $i \preceq k$.

[8] Please see localization time statistics in Section.6.4.1

[9] This probability in the current implementation has been set to 1.0. Please see Section.5.5 for a detailed description on feature vector comparison

Once a candidate list of source code fragments (i.e. statement sequences) has been chosen the actual pattern matching takes place between the chosen starting statement and the outgoing transitions from the current active APM's state. If the type of the abstract statement that the transition points to and the source code statement are compatible (compatibility is computed by examining the Static Model) then feature comparison takes place. This feature comparison is based on the same principles as described in Section.3.2. A similarity measure is established by this comparison between the features of the abstract statement and the features of the source code statement. If composite statements are to be compared, an expansion function "flattens" the structure by decomposing the statement into a sequence of its components. In case of composite statements a nested matching process is initiated.

For example, an `if` statement will be decomposed to a sequence of an `expression` (for its condition), a `then` part and an `else` part. Composite statements generate nested matching sessions similar to the one discussed in the DP-based code-to-code matching.

The process terminates when all possible mathces to reach a final state have been tried. The maximum length sequence of matched statements $S_1; S_2; ..; S_i$ $(i \preceq k)$ that has the maximum matching probability among the sequences of statements of the same length taken from the candidate sequences in a `begin-end` block $\mathcal{B}$, is chosen as the result of the matching process. The Viterbi algorithm guarantees that all possible paths to a final state have been examined, and that the best path (maximizing the overall matching probability) can be chosen.

4. ACL Markov Model Generation

Let T_c be the AST of the code fragment $S_1; S_2; ..; S_k$ and T_a be the AST of the abstract pattern $\mathcal{A} = A_1 op A_2 op A_n$.

A measure of similarity between T_c and T_a is the following probability:

$$P_r(T_c|T_a) = P_r(r_{c_1}, , ...r_{c_i}, ...r_{c_I}|r_{a_1}, ...r_{a_l}, ...r_{a_L}) \qquad (4.1)$$

where,

$$(r_{c_1}, ...r_{c_i}, ...r_{c_I}) \qquad (4.2)$$

is the sequence of the grammar rules used for generating T_c and

$$(r_{a_1}, ...r_{a_l}, ...r_{a_L}) \qquad (4.3)$$

is the sequence of rules used for generating T_a. The probability in (4.1) cannot be estimated in practice, because the number of combinations grows exponentially with the number of rules in the sequence [**Corazza90**]. An approximation of (4.1) is thus introduced.

Let $S_1;..;S_k$ be a sequence of program statements. Let $\mathcal{A}$, be an ACL pattern. During the parsing of an ACL pattern $\mathcal{A}$ that generates T_a, an automaton called Abstract Pattern Model (APM)is built containing as states the abstract descriptions $A_1, A_2, ...A_j, A_{j+1}, ..., A_n$. The APM is constructed by following the standard rules of transforming a regular expression to a Finite State Automaton [**Hopcroft79**].

Nodes in the APM correspond to Abstract ACL Statements and arcs represent transitions implementing thus the control flow imposed by ACLs operators contained in pattern $\mathcal{A}$ (i.e. sequencing, concatenation, choice). Each APM node A_j is considered as a Markov source and is replaced by a static, permanently available Markov model whose states are labeled by symbols M_j, called a Source Code Model (SCM) (Fig.5.2). Each node in SCM is used to generate (match) source code.

The Source Code Model is an alternative way to represent the syntax of a language entity and the correspondence of Abstract Statements in ACL with source code statements.

For example, a transition in APM labeled as (pointing to) an `Abstract Iterative Statement` is linked with the `While`, `For` and `Do-While` static model.

Let T'_c be the AST of a sequence of source code statements $S_1;S_2;..;S_i$ $(i \preceq k)$ taken from a candidate sequence of statements $S_1;S_2;..;S_k$

The best alignment between a sequence of statements $\mathcal{S} = S_1;S_2;..;S_i$ and a pattern $\mathcal{A} = A_1;A_2;.....A_j$ is computed by the Viterbi [**Vite67**] dynamic programming algorithm using a feature vector comparison function for evaluating the following approximation of (4.1) [**Brown92**]:

$$P_r(T'_c|T_a) \simeq P_r(S_1;..;S_i|A_1 op..op A_n) = P_r(S_1;..;S_i|\text{ APM }(A_1 op..op A_n)) \tag{4.4}$$

The desired probability $p = P_r(T'_c|T_a)$ is approximated by the result of the application of the Viterbi algorithm to the Markov model:

$$p \simeq P_r(S_1;..;S_i|A_1 op..op A_n) = P_r(S_1;..;S_i|APM(A_1 op..op A_n)) =$$
$$max P_r(S_1;S_2..S_{i-1}|history, A_{reach(i-1)}) \cdot P_r(S_i|history, A_{reach(i)})) \tag{4.5}$$

where,

- *history* gives the sequence of already successfully achieved matches from previous steps,

- *reach(m)* is a function that determines the APM states that can are valid to be considered during the m*th* step. Such function is computed by examining the APM and its corresponding transitions and,
- *reach(i)* represents a final state in the APM generated by the pattern $A_1 op..op A_n$.

The process returns the longest source code statement subsequence $S_1; S_2; ..; S_i$, obtained from the candidate sequence $S_1; S_2; ..; S_k$ and has the maximum matching probability among the alternative matches of the same length.

The dynamic model (APM) that has been generated by parsing the given pattern guarantees that only the allowable sequences of comparisons are considered at every step.

The final similarity between a sequence of source code statements and a pattern is given as the magnitude of logarithm of the probability p.

As it is illustrated in (4.5) the final value of p is computed in terms of calculating matching similarity measures in terms of matching probabilities between individual abstract statements and code fragments. This matching similarity measures take the form of the probability value given by $P_r(S_i|history, A_{reach(i)})$. These matching probabilities between individual abstract statements and code fragments are computed as follows:

$$P_r(S_i|history, A_{reach(i)}) = P_{scm}(S_i|A_{reach(i)}) \cdot P_{comp}(S_i|history, A_{reach(i)}) \tag{4.6}$$

where,

- $P_{scm}(S_i|A_{reach(i)})$ is the probability that the Static Markov Model generates statement S_i given the description $A_{reach(i)}$
- $P_{comp}(S_i|history, A_{reach(i)})$ is the probability computed by comparing the feature vectors between S_i and $A_{reach(i)}$, given the matching history (i.e. existing bindings).

The feature vector comparison function is discussed in the following subsection.

As an example consider the APM in Fig.5.2 generated by the pattern $A_1; A_2^*; A_3^*$, where A_j is one of the legal statements in ACL. The Viterbi algorithm applied to this model for a selected candidate code fragment $S_1; S_2; S_3; S_4$ provides the best path (maximizes the matching probability) for matching this code fragment with the given model. The comparisons that take place for the given example APM and the code fragment are illustrated in Fig.5.3. This example illustrates the application of the Viterbi algorithm and the matching process between the source code statements $S_1; S_2; S_3; S_4$ and the ACL pattern $A1; A2^*; A3^*$.

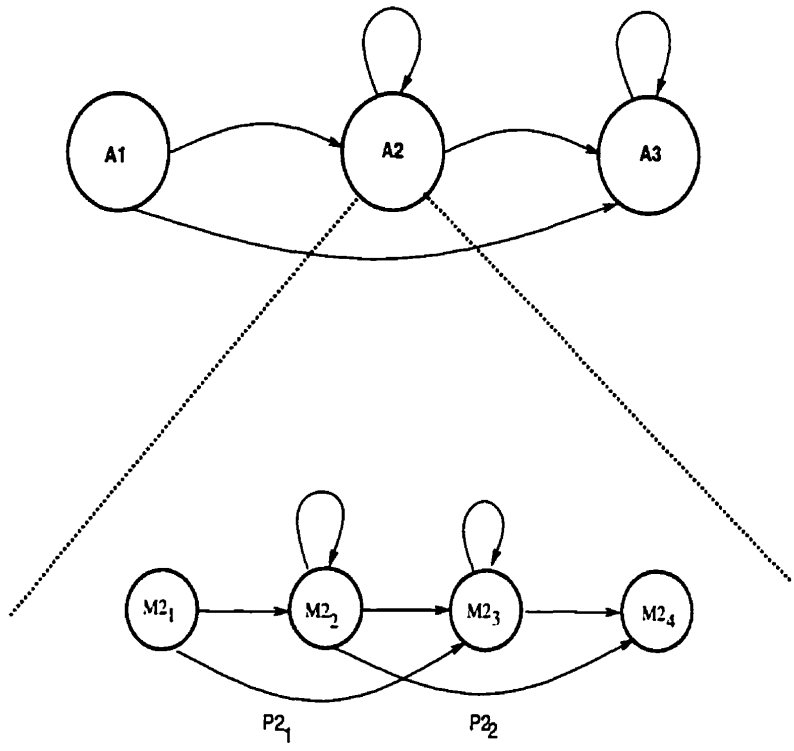


FIGURE 5.2. A dynamic model for the pattern $A1; A2^*; A3^*$

The Viterbi algorithm proceeds by selecting always the path that maximizes the overall probability. In the case of position (S_3, A_3^*) in Fig.5.3, two incoming selections are possible; one (horizontal transition) from position (S_2, A_3^*) that corresponds to the probability $P_r(S_1; S_2|A_1, A_3^*)$ and another (diagonal transition) from position (S_2, A_2^*) that corresponds to the probability $P_r(S_1; S_2|A_1, A_2^*)$. Based on these values and the computed values of P_{scm} and P_{comp} the Viterbi algorithm chooses the best incoming path and updates *history* accordingly.

Note that at every step, the probabilities of the previous steps are stored and there is no need to be reevaluated.

For example $P_r(S_1, S_2|history, A_2^*)$ is computed in terms of $P_r(S_1|A_1)$ which is available from the previous step.

With each transition in the Static Model a list of probabilities based on the type of expression likely to be found in the code for the plan that we consider is attached.

An example of a static model for the `abstract pattern-expression` is given in Fig. 5.4. Here we assume for simplicity that only four C expressions can be generated by a `abstract pattern-expression`.

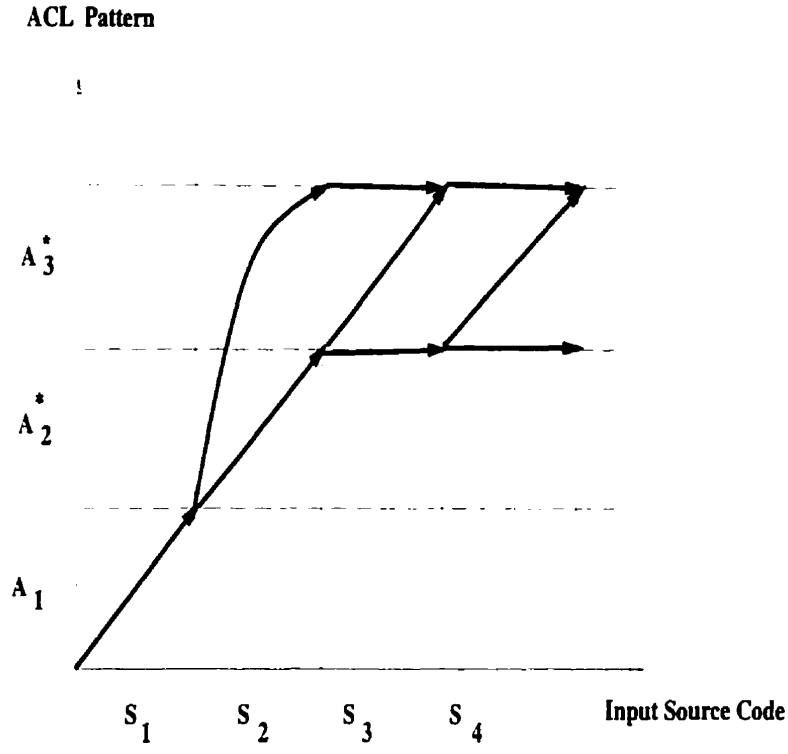


FIGURE 5.3. Dynamic Programming driven comparisons between an ACL pattern $A_1; A_2^*; A_3^*$, and a code fragment $S_1; S_2; S_3; S_4$

For example, in the `Traversal of a linked list` plan the `while` loop condition, which is an expression, most probably generates an `inequality` of the form *(list-node-ptr != NULL)* which contains an identifier reference and the keyword NULL.

A crucial problem with this approach is the estimation of these probabilities for the HMMs. Initially, probabilities can be established subjectively and modified as far as new data are fed.

The initial probabilities in the static model are provided either:

- by the system giving default values based on a uniform distribution in all outgoing transitions for a given state as in Fig.5.4 or,
- by the user who may provide some subjectively estimated values while he or she is formulating the query using the ACL primitives. These values may come from the knowledge that a given plan is implemented in a specific way. In the above mentioned example of the `Traversal of a linked list` plan the `Iterative-Statement` pattern usually is implemented with a `while` loop. In such a scenario the `Iterative` abstract statement can be considered to generate a `while` statement with higher probability than a `for` statement. Similarly, the expression in the *while* loop is more likely to be an *inequality* (Fig. 5.4). Once the system is used and results are evaluated these probabilities can be adjusted to improve the performance.

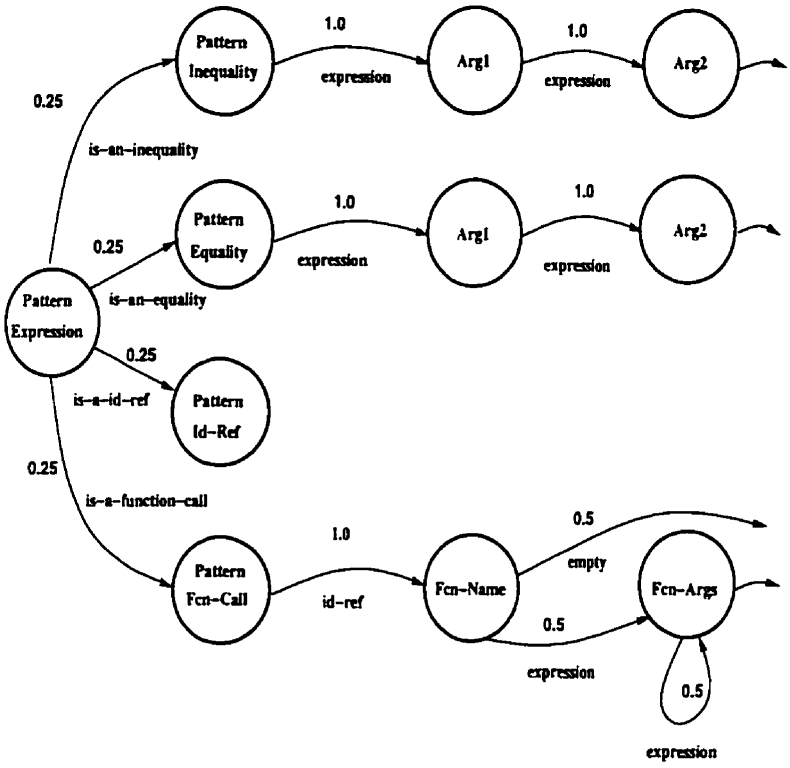


FIGURE 5.4. The static model for the expression-pattern. Different transition probability values may be set by the user for different plans. For example the *traversal of linked-list* plan may have higher probability attached to the *is-an-inequality* transition as the programmer expects a pattern of the form *(field != NULL)*

Probabilities can be dynamically adapted to a specific software system using a cache memory method originally proposed (for a different application) in [**Kuhn90**]. A cache is used to maintain the counts for most frequently recurring statement patterns in the code being examined. Static probabilities can be weighted with dynamically estimated ones as follows :

$$P_{scm}(S_i|A_j) = \lambda \cdot P_{cache}(S_i|A_j) + (1 - \lambda) \cdot P_{static}(S_i|A_j) \qquad (4.7)$$

In this formula $P_{cache}(S_i|A_j)$ represents the frequency that A_j generates S_i in the code examined at run time while $P_{static}(S_i|A_j)$ represents the a-priori probability of A_j generating S_i given by the static model SCM. λ is a weighting factor. The choice of the weighting factor λ indicates the user's preference on what weight he or she wants to give to the feature vector comparison. Higher λ values indicate a stronger preference to depend on what has been matched so far (i.e. the programming style for a plan). This preference gives a "local" view to the matching process. Lower λ values indicate preference to match independently of what has been matched so far. This preference gives

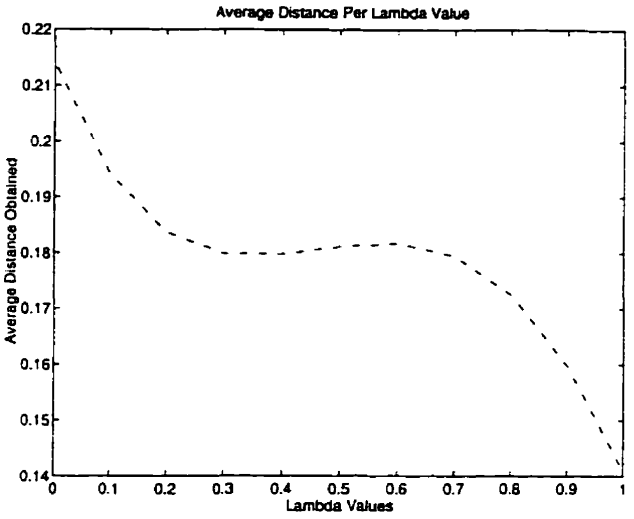


FIGURE 5.5. Effect of λ values to final probability calculation

a "global" view to the matching process as any source code statement can be generated using a more uniform probability distribution, given a-priori by the system. The values of the final reported average distance ($-log(P_{scm}(S_i|A_j)$) between a query and all the successful code matches as a function of different values of the co-efficient λ are shown in Fig.5.5. Fig.5.5 suggests that:

- There is a set of values (in this example 0.3-0.6) for which the distance remains at similar levels (i.e. λ balances the effects of P_{cache} and P_{static}).
- For the rest of the values of the co-efficient λ, we observe that the higher its value is the lower the computed distance (i.e. the higher the matching probability is). This is an accommodating result as it suggests that P_{cache} can be used for tailoring the matching process to a particular "programming style" evident in the software system that is being analyzed.

The value of λ can be computed by deleted-interpolation as suggested in [**Kuhn90**]. As proposed in [**Kuhn90**], different cache memories can be introduced, one for each A_j. We use different cache memories for each A_j, and we compute the value of λ to be proportional to the amount of data stored in each cache.

5. Feature Vector Comparison

In this section we discuss the mechanism used for calculating the similarity between two feature vectors. Note that S_i's and A_j's feature vectors are represented as annotations in the corresponding ASTs.

The feature vector comparison of S_i and A_j returns a value $p = P_r(S_i|A_j)$.

The features used for comparing two entities (source and abstract) are:

(i) *Variables defined* $\mathcal{D}$: Source-Entity $\rightarrow$ {Identifier} [10]

(ii) *Variables used* $\mathcal{U}$: Source-Entity $\rightarrow$ {Identifier}

(iii) *Keywords* $\mathcal{K}$: Source-Entity $\rightarrow$ {String}

(iv) *Metrics*

- Fan out $\mathcal{M}_1$: Source-Entity $\rightarrow$ Real
- D-Complexity $\mathcal{M}_2$: Source-Entity $\rightarrow$ Real
- McCabe $\mathcal{M}_3$: Source-Entity $\rightarrow$ Real
- Albrecht $\mathcal{M}_4$: Source-Entity $\rightarrow$ Real
- Kafura $\mathcal{M}_5$: Source-Entity $\rightarrow$ Real

These features are AST annotations and are implemented as mappings from an AST node to a set of AST nodes, set of Strings or set of Numbers.

Let S_i be a source code *statement* or *expression* in program $\mathcal{C}$ and A_j an *abstract statement* or *expression* in pattern $\mathcal{A}$. Let the feature vector associated with S_i be $\mathcal{V}_i$ and the feature vector associated with A_j be $\mathcal{V}_j$. Within this framework we used the Jaccard's coefficient considered in the computation as a probability:

$$P_r(S_i|A_j) = \frac{1}{v} \cdot \sum_{n=1}^{v} \frac{card(AbstractFeature_{j,n} \cap CodeFeature_{i,n})}{card(AbstractFeature_{j,n} \cup CodeFeature_{i,n})} \qquad (5.1)$$

where v is the size of the feature vector, or in other words how many features are used, $CodeFeature_{i,n}$ is the n*th* feature of source statement S_i and, $AbstractFeature_{j,n}$ is the n*th* feature of the ACL statement A_j.

As in the code to code dynamic programming matching, lexicographical distances between Identifier names and numerical distances between metrics are used. Within this context two strings are considered similar if their lexicographical distance is less than a selected threshold, and the comparison of an abstract entity with a code entity is valid if their corresponding metric values are less than a given threshold.

Thus, ACL is viewed more as a query language where new features and new requirements can be added and be considered for the matching process. For example a new feature may be a link or invocation to another pattern matcher (i.e. SCRUPLE) [**Paul94**] so that the abstract pattern in ACL succeeds in matching a source code entity if the additional pattern matcher succeeds *and* the rest of the feature vectors match.

[10] Following the REFINE practice, we distinguish between an identifier's introduction (Identifier or Variable) and its occurrence in the source (Identifier-Reference or Variable-Reference). An identifier object is used to introduce a name, which can be a variable, function, etc.. It has a declaration, and its scope is determined by the procedure that it was declared in. An identifier also has a set of occurrences (Identifier-References) associated with it, and may be thought of as a symbol table entry having information about the name.

This is an important point as we do not propose ACL as a specification language. Within the context of this thesis, ACL is a formalism to facilitate pattern matching representing a number of control and data flow properties of a code fragment. ACL can not capture semantics or hidden data and control flow dependencies (i.e. aliasing).

6. Recognition Space

In the sections above, we have defined ACL and the matching process based on Markov-models. ACL patterns can be seen as structural abstractions of source code fragments. In this section we discuss constraints in the space of possible solutions generated by an ACL pattern and investigate relationships between ACL patterns by defining a partial ordering between them. Moreover, we show that for a given a set of implementations of a given algorithm, we can compose a pattern out of the known patterns that generate these possible implementations and is minimal in terms of the source code entities it generates (i.e. generates all implementations and minimal noise).

Definition : An atomic pattern is a pattern that does not contain any other patterns.

Example :

- P_1

```
assignment-statement
   uses    : [?x : * struct]
   defines : [?y : * struct]
```

- P_2

```
function-call ?fcnName : * struct
   uses    : [?param : * char]
```

Definition : A composite pattern is a pattern that is made by other patterns using grouping ($\{\}$), the operators ($\oplus, \|, ;$), and macro inclusion.

Example :

- P_1:

```
if-statement(any-cond) then
    {
        assignment-statement
          uses    : [?x : * struct]
          defines : [?y : * struct]
    }
```

- P_2:

```
(assignment-statement
   defines : [i]) +
(assignment-statement
   defines : [j],
   uses : [k])
```

Definition : A *well identified* pattern is a pattern that matches a recognizable implementation of a particular algorithm in a software system.

Example : Suppose the algorithm (plan) is to update a counter when an element is found. We can classify this plan as an *update-counter-on-condition* plan. A pattern that matches an implementation of this plan is:

```
update-counter-plan(?target, ?elem, ?count) =
   if-statement(equality
                uses : [?target, ?elem] ) then
      {
         *-statement;
         assignment-statement
           uses : [?count      : numeral],
           defines : [?count   : numeral];
         *-statement
       }
```

The pattern above matches the code:

```
if (temp_var->name == name)
 {
  cl_print("werror","\nFact address ?");
  cl_print("werror",symbol_string(name));
  cl_print("werror"," also used as variable name\n");
  count_flag ++;
 }
```

Definition : An hierarchical pattern is a composite pattern that is built by using well identified patterns.

Example : Suppose the algorithm is to count the occurrences of a particular element in a linked list. A pattern that matches an implementation of this algorithm is:

```
count-occurrences-of-an-elem-in-linked-list(?head, ?ptr, ?target, ?count) =
 1          actual-assignment-statement
 2              uses    : [?head : * struct ],
 3              defines : [?ptr : * struct];
 4          *-statement;
 5          while-statement(inequality
 6                          uses : [?ptr : * struct])
 7               {
 8                  *-statement;
 9                  if-statement(equality
10                                  uses : [?target, ?ptr : * struct] )
11                    then
12                       {
13                         *-statement;
14                         assignment-statement
15                              uses : [?count : numeral],
16                              defines : [?count : numeral];
17                         *-statement
18                       };
19                  *-statement
20                          abstract-expression-description
21                           empty
22               }
```

The pattern above contains from lines 9-18 the well-identified plan *update-counter-on-condition(?target, ?ptr, ?count)*. Consequently the pattern can be rewritten as :

```
count-occurrences-of-an-elem-in-linked-list(?head, ?ptr, ?target, ?count) =
            actual-assignment-statement
                uses    : [?head : * struct ],
                defines : [?ptr : * struct];
            *-statement;
            while-statement(inequality
                        uses     : [?ptr : * struct])
                 {
                    *-statement;
```

```
                update-counter-on-condition(?ptr, ?target, ?count);
                *-statement
            }
```

Definition : A template pattern is a potential hierarchical pattern that is parameterized on the plans it may include.

Example : In several cases, generic patterns can be enhanced by the inclusion of other patterns to form more specific ones. For example, the plan *traverse-a-linked-list-and-do* can be specialized by including more specific actions in the body of the while-statement in the ACL pattern below

```
traverse-a-linked-list-and-do(?head, ?currPtr)<T> =
        actual-assignment-statement
            uses    : [?head : * struct ],
            defines : [?ptr : * struct];
        *-statement;
        while-statement(inequality
                    uses : [?ptr : * struct])
            {
                *-statement;
                <T>;
                *-statement
            }
```

Here T denotes a well identified pattern that can be included in the generic pattern to form a more specific pattern (i.e. a specialization).

For example, when the template T is instantiated with the plan *update-counter-on-condition(?ptr, ?target, ?count)* then the plan *count-occurrences-of-an-elem-in-linked-list* is formed which can be written as:

```
count-occurrences-of-an-elem-in-linked-list(?head, ?ptr, ?target, ?count) =
    traverse-a-linked-list-an-do(?head, ?ptr)
            <update-counter-on-condition(?ptr, ?target, ?count)>
```

Inclusion works well when a complex plan can be decomposed in a number of distinct simpler plans. When overlapping or scattered plans are involved then simple inclusion may not adequate. In such cases we have to abstract and normalize the representation of the program by using program representation techniques similar to the ones suggested in [**Wills93**], [**Hartman91a**], [**Quilici94**], [**Choi90**], [**Hausler90**]. The framework of this thesis is to provide insights for the pattern matching

part that is computationally expensive in these approaches. One of the future enhancements of the work presented here is to provide formalisms in ACL to represent more abstract concepts such as data flow dependencies, pre and postconditions for each statement and value ranges for the Abstract Identifiers.

Definition : Let P be a pattern represented in ACL. We define *Coverage* of P $Cov(P)$ be the set of code fragments that is generated by P. More formally,

$$Cov(P) = \{\text{S s.t. S} = [S_1; S_2; ..S_k] \text{ and } p(S_i|P) \geq \text{similarity threshold}\}$$

Example :

For the *linked-list-count-elements* related pattern when is applied to the CLIPS program a part of its *Coverage* is:

FILE	FROM-LINE	TO-LINE
"deffacts"	515	518
"intrfile"	721	722
"intrfile"	732	736
"memory"	87	88
"memory"	95	96
"method"	680	689
"method"	1687	1713

Definition : Two patterns P_1, P_2 are equal within the context of a software system S

$$(P_1 =_S P_2) \quad \textit{iff} \quad Cov(P_1) = Cov(P_2).$$

Definition: Given a set S, a partition Π of S, is a collection of subsets S_k of S with the properties

- $S_i \cap S_j = 0$ if $i \neq j$ (the S_k are disjoint)
- $\cup S_k = S$ (they exhaust S).

That is, each element $x \in S$ is in one and only one S_k, and thus Π decomposes S in various subparts.

Lemma : Let $\mathcal{R}$ be the set of all source code statement sequences that can be taken from the software system S. Any pattern P partitions $\mathcal{R}$.

This is straightforward to prove if we observe that P either:

(i) generates (matches) a set of source code statement sequences and $Cov(P) \neq \emptyset$.

(ii) does not generate any source code statement in S, that is, $Cov(P) = \emptyset$.

In the first case P partitions $\mathcal{R}$ in two sets R_1 and R_2. The first set is the set of statement sequences that can be generated by P that is, $Cov(P)$, The second set consists of code sequences that are not matched by P. Apparently, the two sets are disjoint and exhaust $\mathcal{R}$.

In the second case the partition is trivial and is composed of the set (R) itself.

Definition : Given a source code fragment C we define *abstraction* of C be a set of patterns $\mathcal{P} = \{P_1, ...P_k\}$ such that for every $P_i \in \mathcal{P}$, $C \in Cov(P_i)$

Example : Given the code fragment C_1:

```
while(p != NULL)
     {
      if (!strcmp(p->type,"child"))
         {
          count = 1;
          printf("[%s]\n",p->link->name);
         }
      p = p->next;
     }
```

an abstraction of it are the patterns P_1, P_2 because both match C_1

- P_1 :

```
while-statement(any-cond
                uses : [?ptr : * struct])
  {
    +-statement
  }
```

and

- P_2 :

```
while-statement(any-cond
                uses : [?ptr : * struct])
  {
```

```
    *-statement;
    if-statement(any-cond
                    uses : [?target, ?ptr : * struct] ) then
       {
         *-statement;
         function-call printf
            uses : [?ptr : * struct]
       };
    *-statement
}
```

Definition : Given a source code fragment C and an *abstraction* of it $\mathcal{P} = \{P_1, ...P_k\}$ we define the closest pattern to C in $\mathcal{P}$ be, any pattern $P_j \in \mathcal{P}$ for which

$$p(C|P_j) \preceq p(C|P_i) \forall P_i \in \mathcal{P}$$

that is any pattern that produces the least distance when matched with C. Here, $p(C|P_j)$ denotes the probability that code fragment C is matched by the pattern P_j.

Definition : Given two patterns P_i, P_j, we define P_i is *no more general than* P_j ($P_i \preceq_g P_j$) iff

$$Cov(P_i) \subseteq Cov(P_j)$$

.

Example : In the example patterns given above, $P_2 \preceq_g P_1$ because the *Coverage* of P_1 is :

FILE	FROM-LINE	TO-LINE
"object"	1131	1139
"object"	1283	1291
"object"	2236	2244
"object"	2095	2103
"object"	2261	2272
"object"	2200	2215
"object"	2166	2194
"object"	2121	2132
"object"	2058	2072
"object"	2025	2052

while the *Coverage* of P_2 is :

FILE	FROM-LINE	TO-LINE
"object"	1131	1139
"object"	1283	1291
"object"	2236	2244
"object"	2095	2103

and thus $Cov(P_2) \subseteq Cov(P_1)$.

Theorem: The relation *no more general than* is a partial order relation.

Proof:

- The relation *no more general than* is reflexive ($P \preceq_g P$). This holds because $Cov(P) \subseteq Cov(P)$.
- The relation *no more general than* is antisymmetric ($P_i \preceq_g P_j$ and $P_j \preceq_g P_i$ imply $P_i = P_J$). This holds because if $Cov(P_i) \subseteq Cov(P_j)$ and $Cov(P_j) \subseteq Cove(P_i)$ then $Cov(P_j) = Cov(P_i)$, that is $P_i =_S P_j$.
- The relation *no more general than* is transitive. This holds because if $P_i \preceq_g P_j$ and $P_j \preceq_g P_k$ then $Cov(P_i) \subseteq Cov(P_j)$ and $Cov(P_j) \subseteq Cov(P_k)$. This implies $Cov(P_i) \subseteq Cov(P_k)$ that is $P_i \preceq_g P_k$.

Definition : A *chain* is a set of ACL patterns in which all of its elements are related with the *no more general than* relation.

Definition: Two distinct patterns are *comparable* if $P_1 \preceq_g P_2$ or $P_2 \preceq_g P_1$ otherwise are *incomparable.*

Definition: Given a *chain* $P = \{P_1, P_2, ...P_k\}$, a lower bound of a subset X of P is a pattern $P_i \in P$ such that $P_i \preceq_g P_j \ \forall \ P_j \in X$.

Similarly, an upper bound of a subset X of P is a pattern $P_i \in P$ such that $P_j \preceq_g P_i \ \forall \ P_j \in X$.

Definition: Given a *chain* $P = \{P_1, P_2, ...P_k\}$ then the least upper bound of a subset X of P is a pattern $P_i \in P$ such that P_i is an upper bound for X and, for all upper bounds P_n of X $P_i \preceq_g P_n$. An upper bound of X is a pattern P_i whose $Cov(P_i)$ has the maximum cardinality among

the *Coverages* of all patterns $P_n \in X$. The least upper bound coincides with the upper bound, given the definition of the equality between two patterns.

Similarly, the greatest lower bound of a subset X of P is a pattern $P_i \in P$ such that P_i is a lower bound for X and, for all lower bounds P_n of X $P_n \preceq_g P_i$. A lower bound for X is a pattern P_i whose $Cov(P_i)$ has the minimum cardinality among the *Coverages* of all patterns $P_n \in X$. The greatest lower bound coincides with the lower bound, given the definition of the equality between two patterns.

Lemma : A *chain* is a complete lattice. That is, each subset X of P has a least upper bound and a greatest lower bound.

Proof:

Let $P = \{P_1, P_2, ...P_k\}$ be a chain.

Let $X = \{P_i, P_{i+1}, ...P_m\}$ be a subset of P. Since X is a subset of P then X is a *chain.* If we select the pattern $P_u \in X$ such that $Cov(P_u)$ has the maximum cardinality among all elements of X, then P_u is the least upper bound of X. This holds as all elements of X are associated by the *no more general than* relation and the the fact that $Cov(P_u)$ has the maximum cardinality among all patterns in X, means that $Cov(P_j) \subseteq Cov(P_i)$ which implies that $P_j \preceq_g P_u \; \forall \; P_j \in X$.

Similarly, if we select the pattern $P_l \in X$ such that $Cov(P_l)$ has the minimum cardinality among all elements of X, then P_l is the greatest lower bound of X. This holds as all elements of X are associated by the *no more general than* relation and the the fact that $Cov(P_l)$ has the minimum cardinality hence $P_l \preceq_g P_j \; \forall \; P_j \in X$.

Theorem

Let $\mathcal{A}$ be an algorithm and $\mathcal{I} = \{I_1, I_2, ...I_n\}$ be a set of implementations of the algorithm $\mathcal{A}$ in the software system S.

Given the framework above, for every identified algorithm (plan) in a software system S we can construct a pattern that is the closest pattern with respect to the known implementations of $\mathcal{A}$.

Proof:

For every implementation I_j of $\mathcal{A}$, let $R_j = \{P_{j1}, P_{j2}, ...P_{jm}\}$ be an *abstraction* of I_j containing patterns P_{ji} that generate (match) I_j. In this way, R_j is an *abstraction* of I_j, and thus it contains a pattern L_j that is the closest pattern to I_j in R_j (i.e. it produces the least distance when compared with I_j).

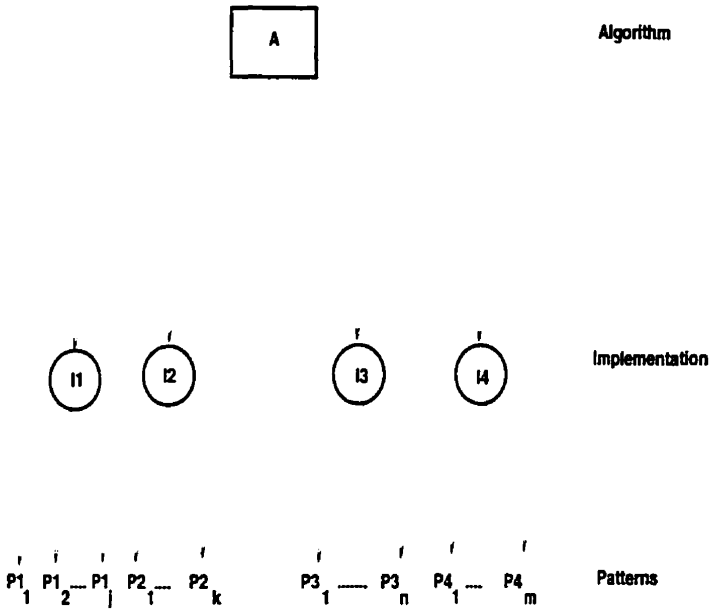


FIGURE 5.6. The algorithm **A** its known implementations I_j in the system, and patterns P_{j_k} that match the implementations.

Let $\mathcal{P}_{\mathcal{A}}$ be $\mathcal{P}_{\mathcal{A}} = L_i .... \oplus L_p$ composed by all the L_i's that do not belong in the same abstraction group R_j. $\mathcal{P}_{\mathcal{A}}$ is the closest pattern with respect to the known implementations of the algorithm $\mathcal{A}$. This is easily proven if we observe that for any known implementation I_j of $\mathcal{A}$ the plan $\mathcal{P}_{\mathcal{A}}$ can match it with minimal distance using the pattern L_j.

Theorem

Let $\mathcal{A}$ be an algorithm and $\mathcal{I} = \{I_1, I_2, ...I_k\}$ be its known implementations in a given software system. Let $\mathcal{P}_1 = \{P_{11}, P_{12}, ..P_{1m}\}, \mathcal{P}_2 = \{P_{21}, P_{22}..P_{2n}\}$ ... $\mathcal{P}_k = \{P_{k1}, P_{k2}, ...P_{kl}\}$ be a collection of ACL patterns that match $I_1, I_2, ...I_k$ respectively (Fig 5.6). Then, we can construct a pattern $P_{min}^{(k)}$ that is composed of known patterns in $\mathcal{P}_i$ (i = 1,2,...k) and is *no more general than* any other pattern P_{comp} that matches all k possible implementations of $\mathcal{A}$ and is also composed of the known patterns. That is

$$P_{min}^{(k)} \preceq_g P_{comp} = (P_{i_j} \oplus ... \oplus P_{q_p})$$

Proof:

Let C be the power-set of $\{I_1, I_2, .. ,I_k\}$ and let this set be sorted in ascending order on the cardinality on its elements. The construction of the pattern $P_{min}^{(k)}$ is a process that involves two phases. During the first phase all patterns that can generate at least *j-many* distinct implementations

are selected. This selection originates either from patterns in $\mathcal{P}_j$, $j = 1, 2, .. ,k$ that may directly generate at least *j-many* distinct implementations or on a $\oplus$ composition of patterns that each generates $m \prec j$ *many* distinct implementations.

The second phase consists of selecting the pattern or the collection of patterns that generate *j-many* implementations with the minimal *Coverage.*

The construction algorithm terminates after k-steps when the pattern that generates *k-many* distinct implementations is computed.

More formally, the construction algorithm operates as follows:

- **Step 0.** Let j $\leftarrow$ 1
- **Step 1.**

  If j = k, then GoTo Step 3, else,

  Select all patterns that generate at least *j-many* distinct elements of all the known implementations of the algorithm $\mathcal{A}$. The *j-many* distinct implementations can be taken from the power-set $\mathcal{C}$. Let $I_j^{(n)}$ denote the *n*th set of *j-many* distinct implementations, taken from $\mathcal{C}$. The selection process is performed by scanning the power set $\mathcal{C}$, and for each element $I_q^{(m)}$ in $\mathcal{C}$, that has cardinality q less than or equal to j (i.e. contains at most j implementations) select those patterns $P \in \{\mathcal{P}_{\ i}$ s.t. i = 1, 2, .. k$\}$ that either

  (i) generate exactly the elements in $I_j^{(n)}$ or

  (ii) when combined with the $\oplus$ operator generate at least all items in $I_j^{(n)}$

  This step produces a number of patterns $P^{(I_j^{(1)})}, P^{(I_j^{(2)})}, ..., P^{(I_j^{(\binom{k}{j})})}$ that generate the $I_j^{(1)}, I_j^{(2)}, .. I_j^{(\binom{k}{j})}$ combinations of *j-many* distinct implementations.

  Let the cost of generating a combination of *j-many* implementations be the minimum cardinality of the *Coverage* among the patterns that generate *j-many* implementations. Let this pattern be denoted as $P_{min}^{(I_j^{(n)})}$. The following step is used to calculate this pattern.
- **Step 2.**

  Select the combination of j-many distinct implementations that can be generated with the least cost from a combination of patterns, and construct the corresponding $P_{min}^{(I_j^{(n)})}$ pattern. This can be done by using dynamic programming.

  Specifically for each $n = 1, 2, 3, ..., \binom{k}{j}$ the Dynamic Programming function that finds the best combination of patterns is :

$$D : Set - of - implementations \rightarrow Real$$

$$D(I_j^{(n)}) = Min \begin{cases} D(I_{j-1}^{(m)}) + D(I_q^{(k)}) \quad and \\ \\ P_{min}^{(I_j^{(n)})} \leftarrow P_{min}^{(I_{j-1}^{(m)})} \oplus P_{min}^{(I_q^{(r)})} \qquad s.t \quad q \prec j \quad and \\ \qquad\qquad\qquad\qquad\qquad\qquad I_j^{(n)} \subset I_{j-1}^{(m)} \cup I_q^{(r)} \\ \\ \\ \\ Cost(P^{(I_j^{(n)})}) \quad and \\ \\ P_{min}^{(I_j^{(n)})} \leftarrow P^{(I_j^{(n)})} \end{cases} \tag{6.1}$$

Where

- $I_j^{(n)}$ denotes the n*th* set of *j-many* distinct implementations, $I_j^{(n)} \in \mathcal{C}$,
- $I_{j-1}^{(m)}$ denotes the m*th* set of *j-1 -many* distinct implementations, $I_{j-1}^{(m)} \in \mathcal{C}$,
- $I_q^{(r)}$ denotes the r*th* set of *q-many* distinct implementations, $q \preceq j$, $I_q^{(k)} \in \mathcal{C}$
- $D(I_j^{(n)})$ denotes the minimal cost for generating the *n*th set of *j-many* implementations
- $Cost(P^{(I_j^{(n)})})$ is the cardinality of $P^{(I_j^{(n)})}$'s *Coverage.*

Let j $\leftarrow$ j + 1 and repeat Step 1.

- **Step 3.**

  Return the pattern $P_{min}^{(k)} \leftarrow P_{min}^{(I_k^{(1)})}$, and its associated cost which is the cardinality of its *Coverage.*

Theorem : The pattern $P_{min}^{(I_k^{(1)})}$ is the least general pattern that matches all implementations $\{I_1, I_2, ..I_k\}$ and is composed of patterns from the elements of the set $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ... \mathcal{P}_k\}$.

Proof :

We will show that pattern $P_{min}^{(I_k^{(1)})}$ is the least general pattern that matches all implementations $\{I_1, I_2, ..I_k\}$ only composed of patterns in the elements of the set $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ... \mathcal{P}_k\}$. We show that by induction on the number of the implementations that we consider in the system.

- **Base Case:** The theorem holds for k = 1.

  When we deal with one implementation *I* only, we can select $P_{min}^{(I_1^{(1)})}$ as the pattern that generates the implementation and has the least *Coverage* from all the other known patterns that match *I*. This pattern is the *glb* of *I*'s *Abstraction.*

- **Induction Assumption:** Suppose that we can construct $P_{min}^{(I_k^{(m)})}$ as the least abstract pattern that matches the m*th* set of *k - many* different known implementations and its solely composed from patterns in elements of $\mathcal{P}$.
- **Induction Step:** We show that the same principle holds for *k+1 - many* known implementations.

  Let $I_{k+1}^{(n)}$ be the n*th* combination of k+1 many implementations taken from the power-set $\mathcal{C}$. The construction algorithm claims that in order to find the pattern that generates $I_{k+1}^{(n)}$ with the least cost (i.e. the *least general* pattern), we have to consider two cases. The first one is based on finding the least cost of generating r-many and q-many implementations where, $q \prec$ k+1 and $r \prec$ k+1. Because of the induction assumption, these quantities can be evaluated and the corresponding patterns be created. In the second case it suffices to pick the pattern that directly generates at least the n*th* combination of the k+1 many implementations from the known patterns in $\mathcal{P}_1, ..\mathcal{P}_k$. This quantity can be evaluated by examining the *Coverage* of each such pattern. Therefore, we can estimate the cost of matching $I_{k+1}^{(n)}$ and construct by the algorithm the pattern $P_{min}^{(I_{k+1}^{(n)})}$.

This result is very accommodating because it ensures the scalability of the pattern-based approach as it allows always to construct abstractions and in particular the least general abstraction from a set of more specific ones.

The ability to build a pattern for more abstract and general algorithms and plans using simpler and more specific ones is fundamental for organizing plan hierarchies, and allowing for hierarchical plan recognition. [11].

[11] Typical examples of hierarchical plan recognition are given in **[Rich90]**, **[Ning94]**

CHAPTER 6

Experiments

In this chapter experiments on the proposed methods are presented together with conclusions that can be drawn from them.

The software systems used for evaluating the proposed approaches are illustrated in Table.6.1. *Clips* is an expert system shell, developed at NASA's Software Technology Center, *tcsh*, and *bash* are popular Unix shells and *Roger* is a real-time speech recognition system developed at McGill University.

The experiments discussed in the chapter fall in the following categories:

(i) Experiments to measure Recall / Precision graphs for the metrics-based method for all possible metric combinations

(ii) Experiments to measure Recall / Precision graphs for the DP-based method and for each program feature used

(iii) Experiments to measure Recall / Precision graphs for the Markov-based method

(iv) Time and Space statistics for measuring the time performance of each matching process and the number of potential clone candidates retrieved.

Precision / Recall graphs, indicating the overall performance of a particular matching process, are based on sample queries involving selected code fragments of different lengths and complexities that expert programmers could identify as components that have been replicated in the system.

Software System	Size (LOC)	# of Files	# of Functions
TCSH	44,754	46	658
CLIPS	32,807	40	705
BASH	27393	63	632
ROGER	13,615	39	235

TABLE 6.1. The Software Systems Used for Experimentation

This selection was necessary in order to be able to calculate Recall values. This type of experiment has been conducted using standard Information Retrieval performance indicator defined below.

Let C be the whole collection of the software components. For each query, the set C can be partitioned into two disjoint sets:

- Q containing R relevant documents and
- Q' containing material irrelevant material.

Similarly, for each such query the set of c-many retrieved elements can be partitioned into two disjoint sets

- q having r relevant documents to the query and
- q' having r' irrelevant documents.

Following the classical Information Retrieval (IR) terminology *Recall* and *Precision* are defined as:

$$Recall = \frac{r}{R} \qquad (1)$$

and

$$Precision = \frac{r}{c} \qquad (2)$$

Important questions such as:

- Size of Corpus,
- How many queries considered and how these were selected,
- How queries were formulated,
- How relevant answers were recognized,
- How non-detected relevant items were computed,

are discussed in the following section.

1. Experimentation Framework

An important point for computing *Recall* and *Precision* is the definition of a measure of *relevance* between a pattern and a retrieved code fragment. To our knowledge, there is no formal definition of *relevance* between two code fragments and there are no standard criteria to recognize one code fragment as being a clone of another. In relevant research studies [**Baker95**], [**Johnson94a**], [**Halst77**],

[McCabe90], [Jankowitz88], code cloning has been seen as a problem of examining statistical, or textual properties of the code. However, experts make fine distinctions on the operations and the criteria for code cloning. Programmers may argue that textual similarity is the most important criterion. Others may argue that the semantics of the system and the Input/Output relations are more important. Within this framework a safe assumption is to:

- use the definitions of code cloning appearing in the literature [Baker95] and,
- obtain feed-back from programmers on establishing the relevant data set for each query. Results obtained for each query are tested against this set to establish Recall, and Precision measurements.

Within this experimentation framework, programmers have identified the following cloning scenaria for two code fragments:

(i) two code fragments they are identical (e.g. are found identical using the Unix utility `diff`), or

(ii) they have the same structure but modified variable names or data types, or

(iii) two code fragments contain common sequences of statements or expressions or,

(iv) one has been obtained by parameterizing the another, or,

(v) one differs from the other on inserted, deleted or substituted statements and expressions.

These scenaria, cover most of the text-based and measurement-based approaches in the current in the literature [Baker95], [Johnson94a], [Paul94]. As far as the semantic-based approaches are concerned, we believe that these can be covered mostly in the framework of language semantics and formal techniques that may indicate functional or behavioral similarity between code fragments. Note that in general, functional and behavioral equivalence is an undecidable problem, and even for the relaxed conditions where we may prove behavioral similarity most of these techniques are not tractable. The ultimate goal of IR is to retrieve components to be presented to a user who makes the final decision on their appropriateness. Furthermore, we feel that the semantics approach exceeds the scope of the pattern-matching based framework proposed in this thesis, and can be left as future work.

Following standard Information Retrieval (IR) practice, consultation with programmers of the subject systems was performed in advance to select code fragments that were replicated in the system. Each replicated component has been tagged by its location (file, length in lines of code) and content (number and type of statements it contains). These "tagged" components form the basis for evaluating query results and thus calculating Recall and Precision.

We have considered a space of 940 functions and we formulated 20 queries in total for each method considered. This ratio corresponds well with the same number of queries per number of documents that has been used in standard Information Retrieval test sets [**Maarek91**], with reported ratios in the range of 2.3% to 2.6% have been reported.

Queries fall in two categories:

(i) Queries for code-to-code matching tested on the metrics-based and the DP-based approach

(ii) Queries for the Markov-based approach

The first query type was selected by programmers that have experience with the structure and the contents of the subject systems. Essentially, these queries are code fragments for which the programmers knew to be or to have cloned instances in the subject system. Code fragments were selected based on:

- The knowledge that these were replicated components,
- Coverage of the cloning scenaria discussed above.

The second query type was selected by considering pseudo-code descriptions given by the programmers and correspond to the code fragments considered in the code-to-code matching queries.

These pseudo code queries aimed at testing both the indexing capabilities of the features proposed and the query capabilities of the ACL language itself.

Obtained results were checked against the set of the relevant components that have been identified by the programmers.

The relationship between Recall and Precision has been computed using the standard IR approach which consists of :

- Evaluating Recall and Precision for each query at every given cut-off point,
- Performing macro-averaging so as to obtain a single Recall and Precision value for every given cut-off point,
- Using linear interpolation to obtain Precision values for Recall values that were not effectively achieved.

Linear interpolation was used to compute Precision values p^* for standard Recall values r^* by applying the following formula :

$$p^* = p_1 + \frac{r^* - r_1}{r_2 - r_1}(p_2 - p_1) \qquad (1.1)$$

where r_1, and r_2 are the recall values immediately to the left and to the right of r^* and p_1, p_2 are the corresponding precision values.

2. Metrics-based Matching Experiments

2.1. Precision Per Metric Usage at Max. Recall Level. This experiment illustrates the Precision [1] variation per metric combination used, at Recall level 95.8% (which was the highest effectively achieved Recall level in these experiments), and similarity threshold distance 0.0.

- Using One Metric: among the single metric usage scenario the Precision is higher when using the Kafura metric (Fig.6.1). An explanation for this behavior is the complexity of the metric in terms of the variety of the features used to compute it. Note that in absolute percent Precision values illustrated in Fig.6.1, the Kafura metric achieves a 1.7% Precision.
- Using Two Metrics: the Precision increases when using the combination of the *Kafura* metric with *S-Complexity* (Fig.6.2). An explanation is the reduction in noise introduced by intersecting the *Fanout* feature in both metrics. An interesting point here is the influence of the McCabe metric when combined with the *Kafura* metric. The reason for this is that McCabe introduces a new low correlated feature, and namely, the structure of the Control Flow Graph.
- Using Three Metrics: the Precision increases when using a combination of the *Kafura* metric, the *McCabe* metric and, *S-Complexity* (Fig.6.3). This result is expected as it includes the metrics from the first best two metrics combinations. The interesting point though is the high Precision we obtain at high Recall levels by the use of the *D-Complexity* instead of the *McCabe* metric. This result can be explained by the nature of the *D-Complexity* metric, which essentially imposes the common constraint feature, *global variables* to the *Kafura* metric.
- Using Four Metrics: the Precision increases when using a combination of the *Kafura* metric, the *McCabe* metric, the *S-Complexity* metric, and the *D-Complexity* metric (Fig.6.4). This is, again, an expected result as it contains the best combinations from the above mentioned levels. The interesting point here is the possibility of replacing the *D-Complexity* with the *Albrecht* metric. This replacement can be explained by the I/O features (*read operations*, *files opened*) of the *Albrecht* metric. Note that *globals* and *fanout* in *D-Complexity* have already been covered by the *S-Complexity* and *Kafura* metric. At this point the *Albrecht* metric adds new matching features and this is the reason it makes such a high contribution.
- Using Five Metrics: we can achieve an effective Recall level of 95.8% and maintain a Precision level of 10.2% (Fig.6.6). This is not a discouraging result if one considers that in all of our experiments (involving the samples queries, and brute-force comparison between all function

[1] Please note that the Precision values illustrated in the figures in this chapter are given in percentage points

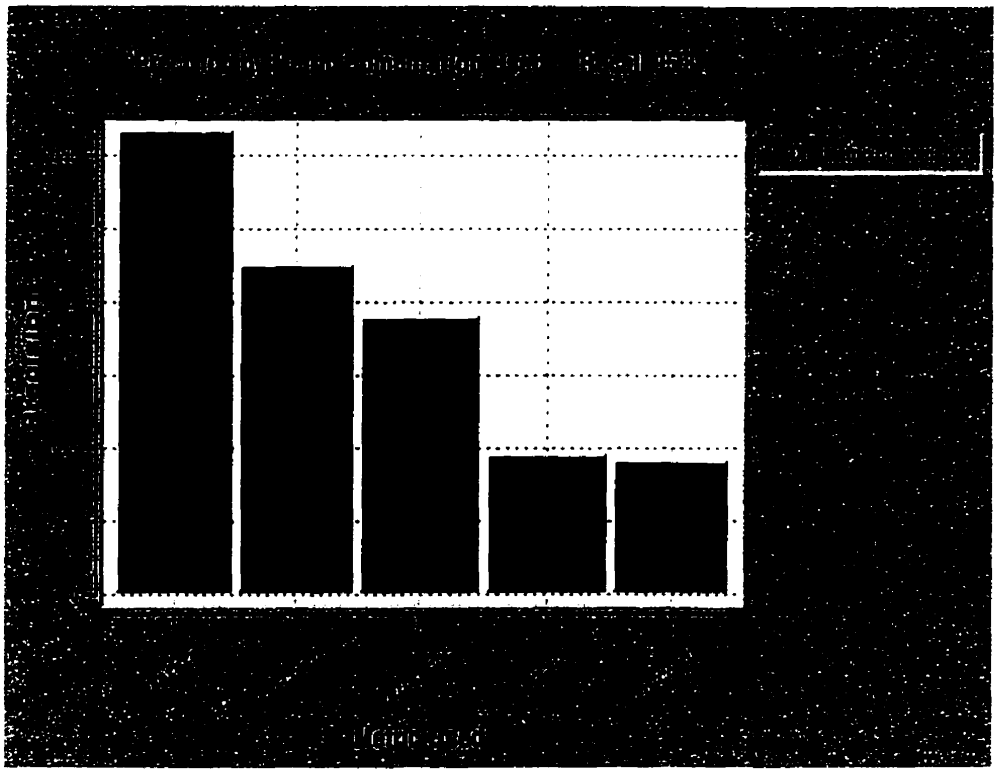


FIGURE 6.1. Precision values (in percentage points) for one Metric used (Recall level 95.8%.)

pairs) we did not retrieve more that 11.3% of the total system size. That means using this approach we can retrieve 11.3% of the system for which we know there are 95.8% of the existing clones.

2.2. Impact of per Metric Threshold Value Variation on Precision. This experiment illustrates the impact of the threshold value changes along every metric axis on precision for a fixed recall value. The motivation for this experiment is to observe the behavior and the impact each metric has on the precision of our results. For this experiment we kept four metrics with a constant zero threshold and varied the fifth metric using the step distance criterion. The impact of threshold

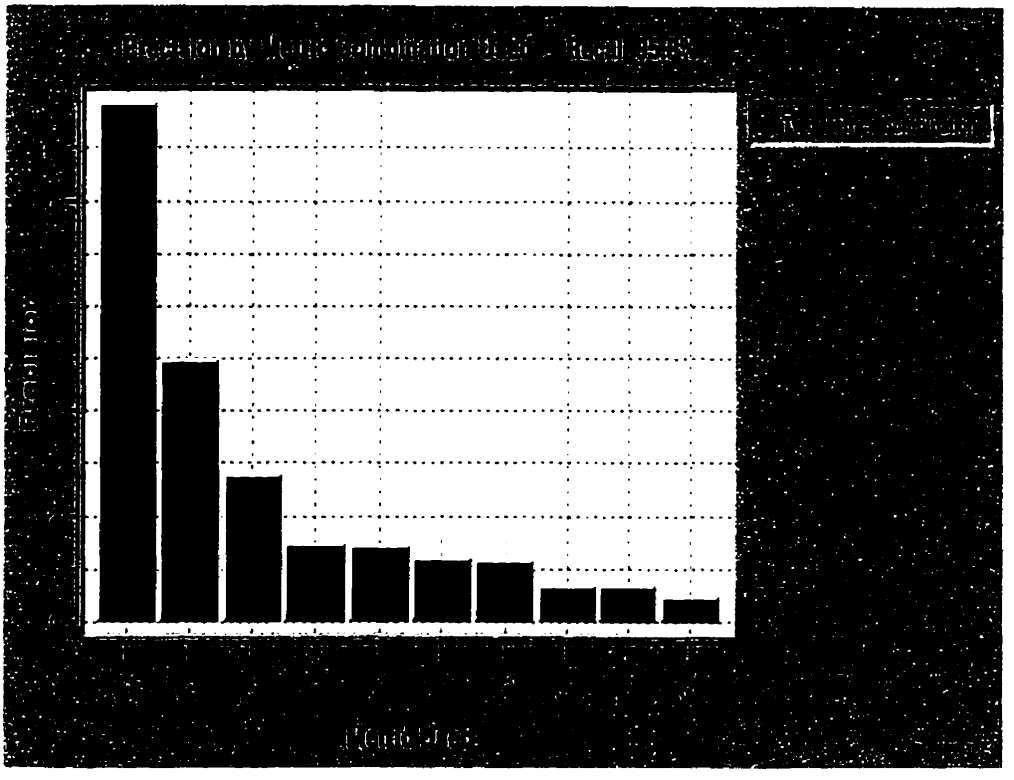


FIGURE 6.2. Precision values (in percentage points) for combinations of two Metrics (Recall level 95.8%.)

changes on the precision of the results is measured as the average percent change of the precision between zero and the 10th step value on each metric axis varied.

The results are illustrated in Fig.6.5 and indicate that the precision is affected the most (drops) for threshold changes in the Albrecht and the Kafura metric. A possible explanation is the complexity and the variety of features used to compute these metrics. By varying a threshold in these metrics, we assume changes in a variety of low correlated features that are not likely to be all changed in a replicated component. Therefore, by increasing the threshold value we allow for more noise to be added which affects Precision. Precisions were measured for 44.4% Recall level.

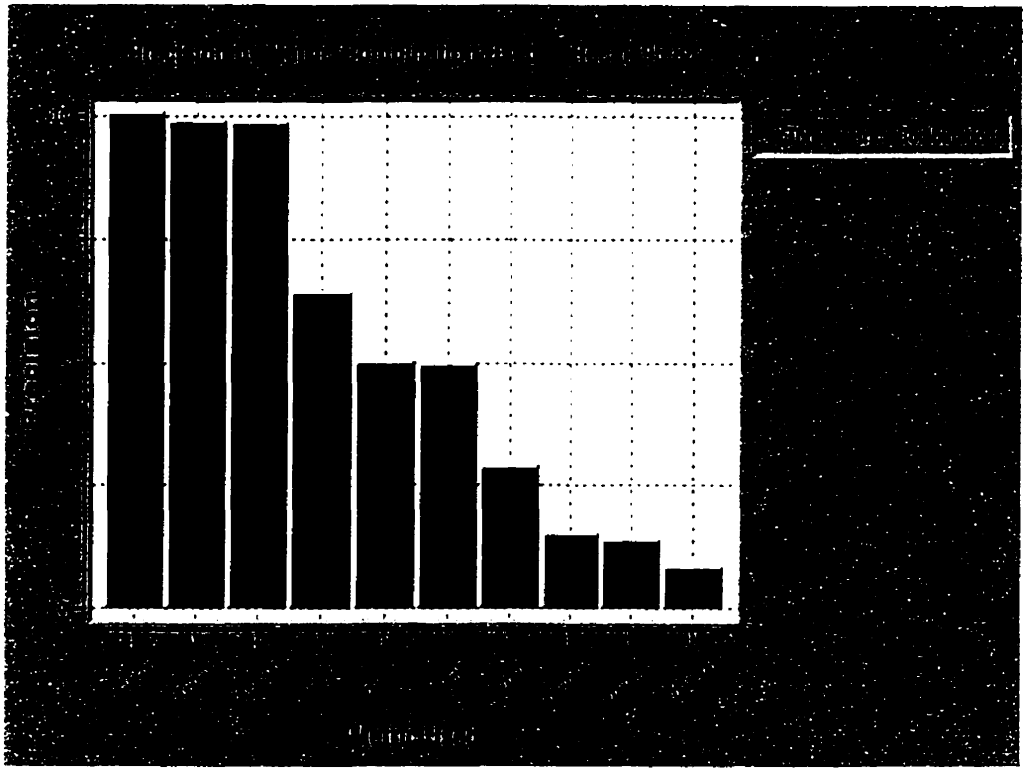


FIGURE 6.3. Precision values (in percentage points) for combinations of three Metrics (Recall level 95.8%.)

2.3. Items Retrieved per Metric Usage. This experiment illustrates the impact of metric usage on the amount of objects retrieved. For this experiment we have used brute force matching between all possible function pairs for each system examined. Values have been averaged for the four systems and for the metric combinations in each category. As expected these results, illustrated in Table.6.2 follow closely the observations on *Precision Per Metric Used for Max. Recall* experiment. However, there is a slight variation in the use of *S-Complexity* and *D-Complexity*. This experiment illustrates that *D-Complexity* can reduce the size of the retrieved components, when combined with the Kafura metric. We believe that a possible explanation for this behavior is

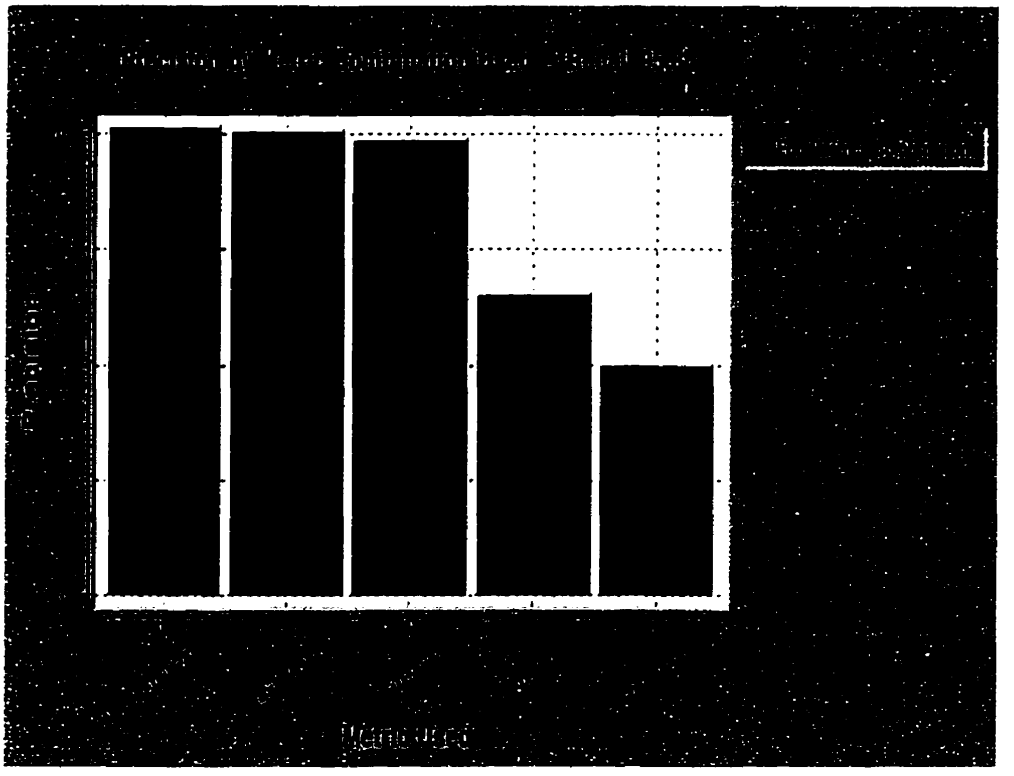


FIGURE 6.4. Precision values (in percentage points) for combinations of four Metrics (Recall level 95.8%.)

the granularity of the components involved in this experiment. This experiment is applied at the function-level, where the impact of data flow may be more evident.

As far as the retrieval times are concerned, note that they are is more sensitive to the number of objects considered during the matching process than to the number of objects finally reported. This explains why similar times are reported for different numbers of pairs reported. Overall, this experiment gives a gross view of the metrics matching behavior.

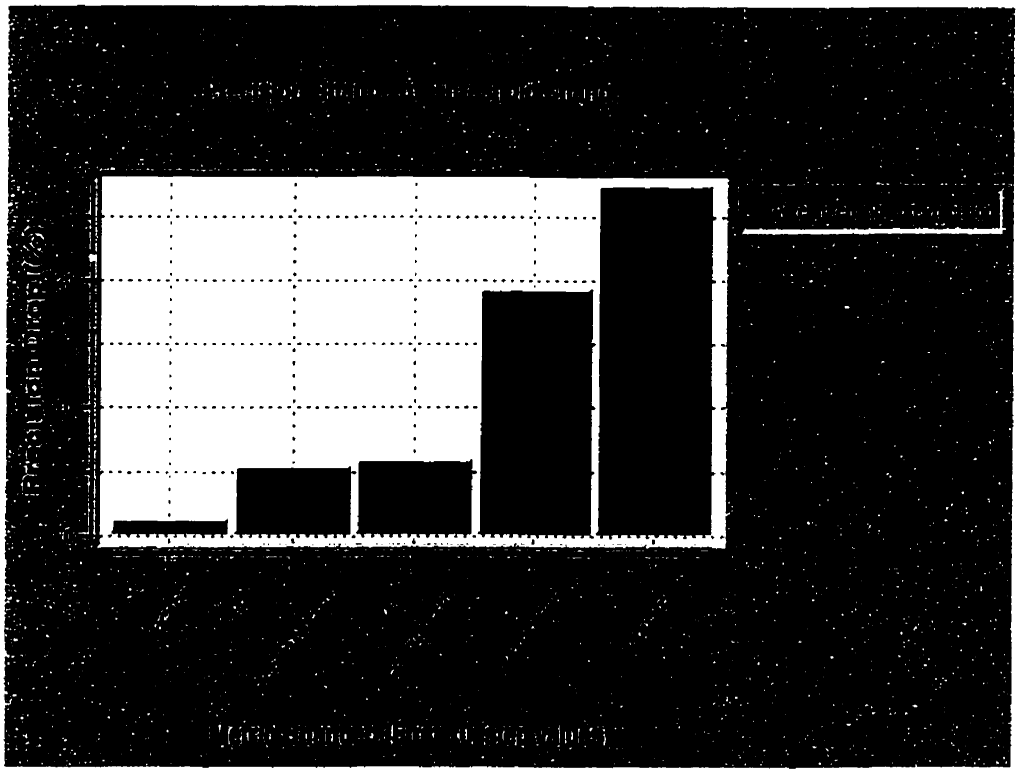


FIGURE 6.5. Precision Change (%) (Drop) by varying threshold values for each metric dimension. Shown is the change between the 1st and the 10th step threshold value.

2.4. Recall Per Distance Range. This experiment illustrates the significance of the distance values to the recognition process. Using all five metrics with threshold set to 2.5 units and cut-off values illustrated in Table.6.3 we measured the recall using our sample queries. The result drawn from this experiment is that using the metrics-based pattern matching technique we obtain most of the clones (57.7%), at distance values $\preceq 0.6$. This result indicates that this technique can be used as a fast first approximation to the clone detection problem. Note that on thresholds set close to zero and using all five metrics this technique is the simplest and fastest to apply. Moreover, the only additional computation involved in this technique is the comparison of the metric values

Metric Combination	*Potential Clone Pairs Retrieved*	*Retrieval Time (Hr:Min:Sec)*
K	1,777	0:14:05
A	14,953	0:42:20
D	15,023	0:42:33
S	17,100	0:54:45
M	26,526	1:40:53
D-K	529	0:09:07
A-K	534	0:11:08
M-K	608	0:12:33
S-K	863	0:11:10
S-D	1,841	0:10:23
S-D	2,365	0:12:57
S-M	3,935	0:12:40
D-M	4.032	0:11:55
M-A	4,100	0:13:30
D-A	12,381	0:17:44
D-M-K	283	0:08:24
M-A-K	288	0:10:47
S-A-K	319	0:08:56
S-D-K	321	0:09:27
S-M-K	339	0:10:00
D-A-K	523	0:10:14
S-M-A	942	0:11:14
S-D-M	1,034	0:10:29
S-D-A	1,837	0:12:01
D-M-A	3,677	0:11:45
S-M-A-K	231	0:09:31
S-D-M-K	231	0:09:31
D-M-A-K	282	0:09:29
S-D-A-K	319	0:10:12
S-D-M-A	942	0:10:15
S-D-M-A-K	231	0:05:27

TABLE 6.2. Metrics-based matching statistics. The size of all possible pairs for this experiment is 248,160. The Recall level for this experiment using all five metrics is estimated as 44.4%.

as the metrics are calculated right after parsing, at link time. Note that the Recall/Precision graph illustrated in the following section suggests for such Recall level a Precision of 46.5% which is a good indication for the usefulness of this technique.

2.5. Recall / Precision. Precision values for specific Recall values were computed by performing an Information Retrieval Experiment. Average Recall values and average Precision values have been used to produce, with linear interpolation, Precision values for standard Recall values (0.0, 0.1, 0.2, .. 1.0). Because of the erratic nature of low recall values for small samples

Distance Range	*Recall*
0.0	44.3%
$\preceq$ 0.2	55.4%
$\preceq$ 0.4	57.7%
$\preceq$ 0.6	57.7%
$\preceq$ 0.8	68.2%
$\preceq$ 1.0	68.2%
$\preceq$ 1.2	82.1%
$\preceq$ 1.4	82.1%
$\preceq$ 1.6	82.1%
$\preceq$ 1.8	82.1%
$\preceq$ 2.5	96.0%

TABLE 6.3. Recall / Distance Value Range (Metrics)

[**Jones81**] we assigned a Precision value of 1.0 to Recall value of 0.0. The collection considered for testing consisted of *Clips*, and the *Roger* system. We formulated 20 queries for which programmers, have identified replicated components in the source code.

In this experiment we report the relation of linear Recall to average Precision, using the best combinations obtained by the experiment discussed in Section 2.1.

The obtained results indicate that:

- Using one Metric (Kafura): we have an almost linear drop in Precision and we obtain a low Precision for high Recall values. This type of analysis does not provide any benefit as all metrics have already been computed and we do not gain in Precision. The matching time using this method is acceptable but is 64% higher than the best time performance we can obtain. The drawback of using only one metric is the drop in Precision which is almost 7 times lower than the best we can achieve for the highest Recall value.
- Using two Metrics (Kafura, S-Complexity): we have a significant gain in Precision for low to medium Recall levels. This can be explained by the common constraint imposed by the *fanout* feature in both metrics. At higher Recall levels we achieved lower values than the ones achieved using the Kafura metric alone. We suspect, though, this is because of the interpolation noise in previous curve above (one metric).
- Using three Metrics (Kafura, S-Complexity, McCabe): we have a new gain in Precision for corresponding Recall values. At this point the McCabe metric is the factor for the Precision increase as it adds the Control Flow component to the already considered features.
- Using Four Metrics(Kafura, D-Complexity, McCabe, S-Complexity): we have a new gain in Precision that can be explained by the common constraints introduced by the common

features used for *D-Complexity*, *S-Complexity*, and the *Kafura* metric, combined with a new feature introduced by the *McCabe* metric.

- Using Five Metrics: we obtain the best curve. This is the best combination to use among the metrics we considered. However, it is very close to the one using four metrics. This may suggest dropping the *Albrecht* metric altogether or replace it with *S-Complexity*. This would be a reasonable idea, but note that the metrics are computed at link time at linear complexity on the AST nodes, and therefore do not constitute a significant computation bottleneck. In any case, each metric has its advantages and it is not a straightforward choice on which metric to ignore. D-Complexity is easier and little bit faster to compute, but Albrecht is sensitive to more features and may be better to use in the long run. Experiments with higher levels of granularity (function level, also shown in Table. 6.2) indicate that when we compare D-Complexity and Albrecht alone, then Albrecht generates fewer candidates.

3. Dynamic Programming Experiments

3.1. Recall Per Distance Range. This experiment, illustrates the significance of the distance values to the retrieval process. Using each of the three features we consider in the DP approach (Set-Uses, Types, Metrics) and cut-off values illustrated in Table.6.4 we measured the recall using our sample queries. The overall result that appears from this experiment is that, by using the DP-based pattern matching technique, the recall is almost uniformly distributed over the distance values. This indicates that this technique is more accurate on measuring differences occurring between two code fragments. Moreover, effective Recall values of 90% obtained with DP were found to correspond to Precision values of 21.1% when using the Set-Uses criterion, 25.5% when using the Types criterion, and 16.8% when using the Metrics criterion (see Table.6.9). Note that for similar Recall value of 90.0%, the metrics-based matching gave Precision level of 9.0%. This is a very strong indication of the usefulness of this approach when the user is willing to invest more time for the matching process in order to gain in precision. The DP-approach is particularly useful for identifying components that have changed between versions of a large system.

Retrieval time performance are illustrated in Table.6.5.

3.2. Precision Per Distance Range (Set-Uses Matching Criterion). This experiment was performed to evaluate the overall Precision behavior of the DP matching using the *Set-Uses* criterion (Fig.6.7). This experiment was performed by applying brute-force matching between all functions in each one of the software systems we used for our experimentation, with similarity threshold set to 0.0. The results were inspected manually to calculate the Precision level.

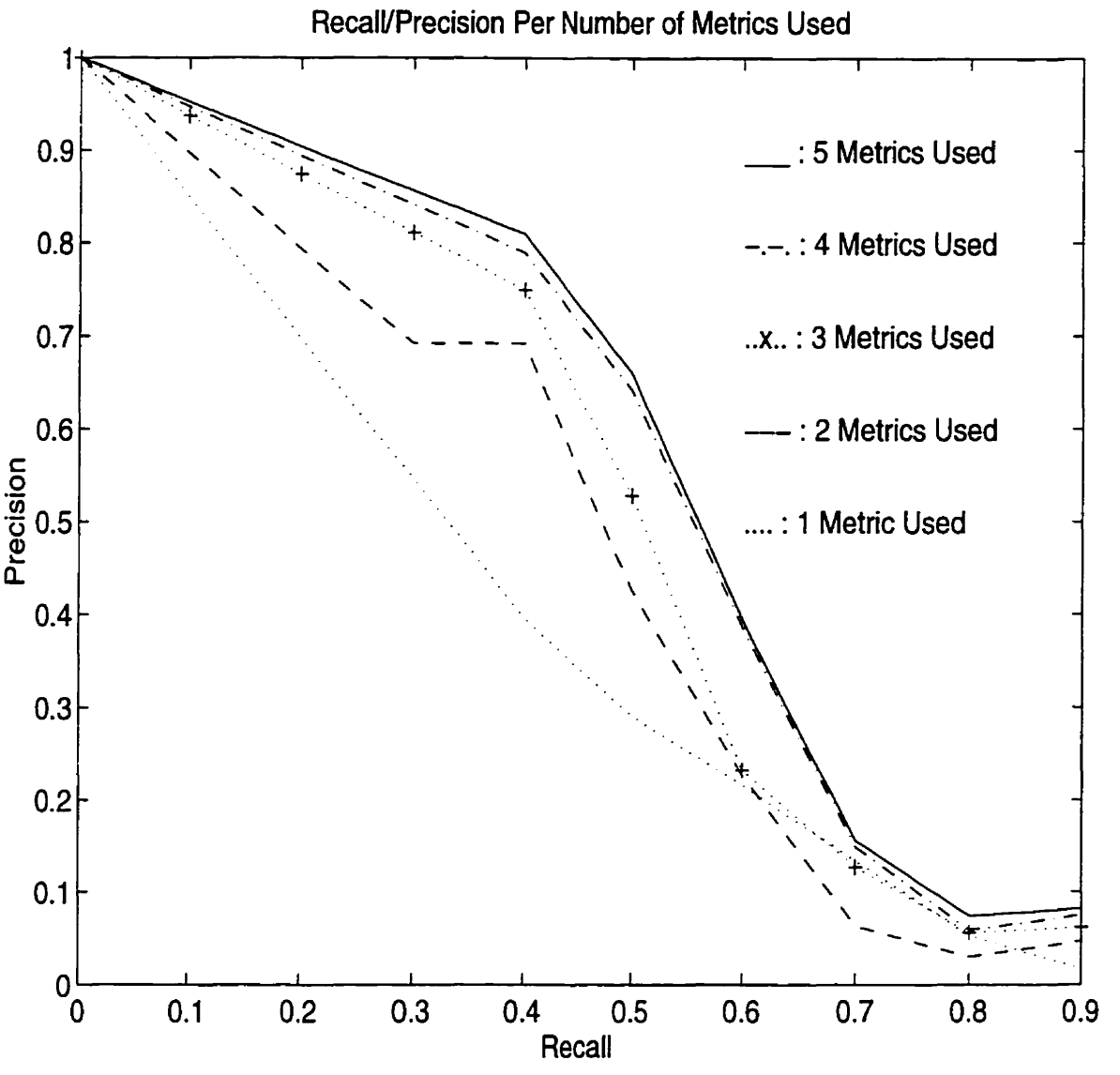


FIGURE 6.6. Precision/Recall Graph for different metric combinations. The metric combinations were selected among the ones that give the highest precision in their category class (i.e. the best combination of two metrics is S-Complexity and Kafura)

The first observation is that DP is a very good method for localizing exact clones. This is indicated by the 100% precision on the obtained results when the threshold has been set to zero. This is of no surprise as by setting the threshold to zero we force the matcher to consider no insertions, nor deletions nor substitutions and require that variable names between the two components have not been altered. These requirements do not leave any margin for considering cases where one code fragment is a modification of the other. The corresponding Recall level was 14.0%. The second observation is that the Precision drops by 17.3% and stays at acceptable levels (82.7%) for values in

Distance Range	*Recall Level (Set-Uses)*	*Recall Level (Metrics)*	*Recall Level (Types)*
0.0	14.0%	39.4%	17.9%
$\preceq$ 0.5	14.0%	39.4%	17.9%
$\preceq$ 0.99	20.4%	44.4%	17.9%
$\preceq$ 1.5	24.4%	44.4%	17.9%
$\preceq$ 1.99	24.4%	56.9%	30.0%
$\preceq$ 2.99	34.4%	58.9%	35.0%
$\preceq$ 3.99	41.4%	58.9%	42.0%
$\preceq$ 4.99	46.4%	58.9%	44.5%
$\preceq$ 7.99	58.9%	74.9%	51.4%
$\preceq$ 11.99	72.1%	92.1%	84.9%
$\preceq$ 15.0	92.1%	92.1%	89.9%

TABLE 6.4. DP-based matching statistics. The size of all possible pairs for this experiment is 248,160. The Recall level achieved for this experiment is estimated as 44.4%.

Criterion Used	*Potential Clone Pairs Retrieved*	*Retrieval Time (Hrs:Min:Sec*
Set-Uses	231	0:12:17
Metrics	231	0:11:02
Types	231	0:08:10

TABLE 6.5. Recall / Distance Value Range (DP)

Distance Range	*Precision Drop (%)*	*Recall Increase (%)*	*Total Recall Achieved (%)*
0.01 - 0.99	-17.3%	+6.4%	20.4%
1.0 - 1.99	-35.1%	+4.0%	24.4%
2.0 - 2.99	-8.8%	+10.0%	30.4%

TABLE 6.6. Recall / Precision Relation Per Distance Value Range (DP Set-Uses Criterion)

the range of 0.01-0.99. This is because such a distance range indicates a small number of changes were needed for two code fragments to be considered clones. The corresponding increase at the Recall level was 6.4%. The next higher drop (by 35.1%) in Precision is introduced for value range from 1.0-1.99 where a Precision of 47.6% is observed. The corresponding gain in Recall is 4.0%. Finally for the distance range of 2.0-2.99, for which we have observed a drop of 8.8% in Precision we have a gain of 10.0% in Recall. These variations are illustrated in Table.6.6. These results indicate that the *Set-Uses* DP-based matching is a useful approach when exact clones are sought, or minor modifications between code fragments are allowed, and the recognition speed is not crucial.

3.3. Precision Per Distance Range (Metrics Matching Criterion). This experiment was performed to evaluate the overall Precision behavior of the DP matching using the *Metrics* criterion (Fig.6.8). The major observation from this experiment is that we can achieve a high

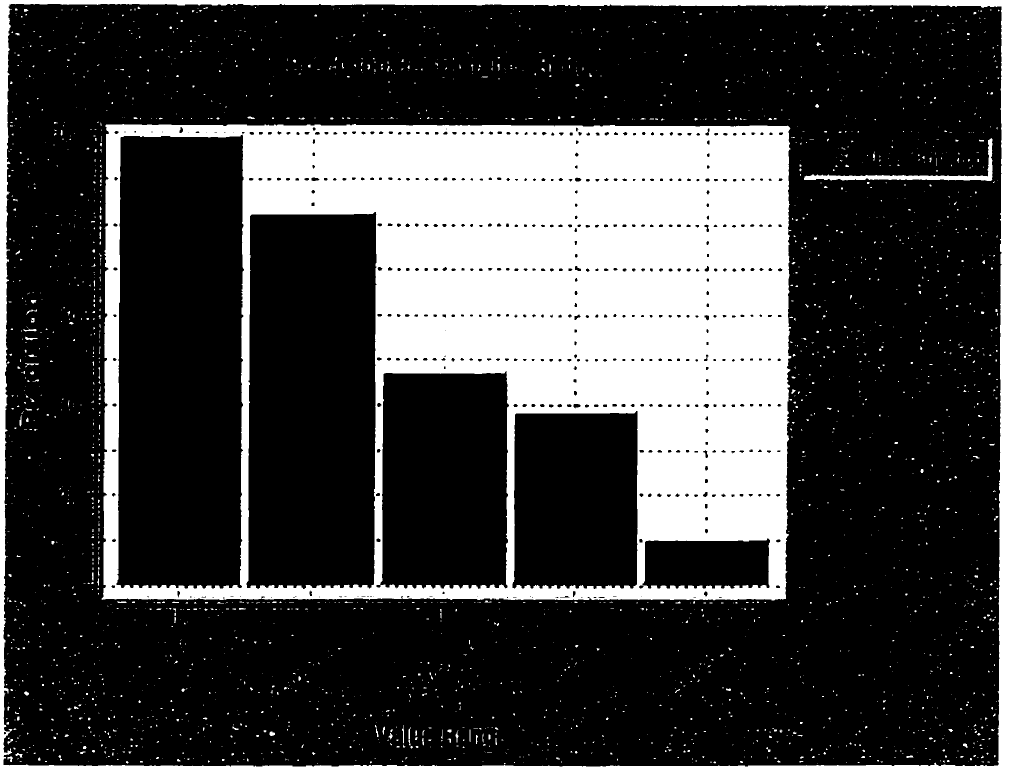


FIGURE 6.7. Average Precision (in percentage points) Per Distance Range for the Set-Uses criterion.

Precision for distances close to zero, but the Precision drops drastically for value ranges where the *Set-Uses* criterion maintained higher Precision values. The reason for this behavior is that the closer the distance is to zero, the fewer modifications between the two code fragments there are and, therefore, the higher the Precision of the achieved results. Once we deviate from distance zero we allow noise, in terms of insertions, deletions and substitutions. Compared to the results obtained using the Set-Uses criterion, two code fragments have a lower distance using the Metrics criterion because metrics when considered at the statement level loose their significance, and, therefore two statements may give a low comparison cost even if they may be different. This has a direct effect

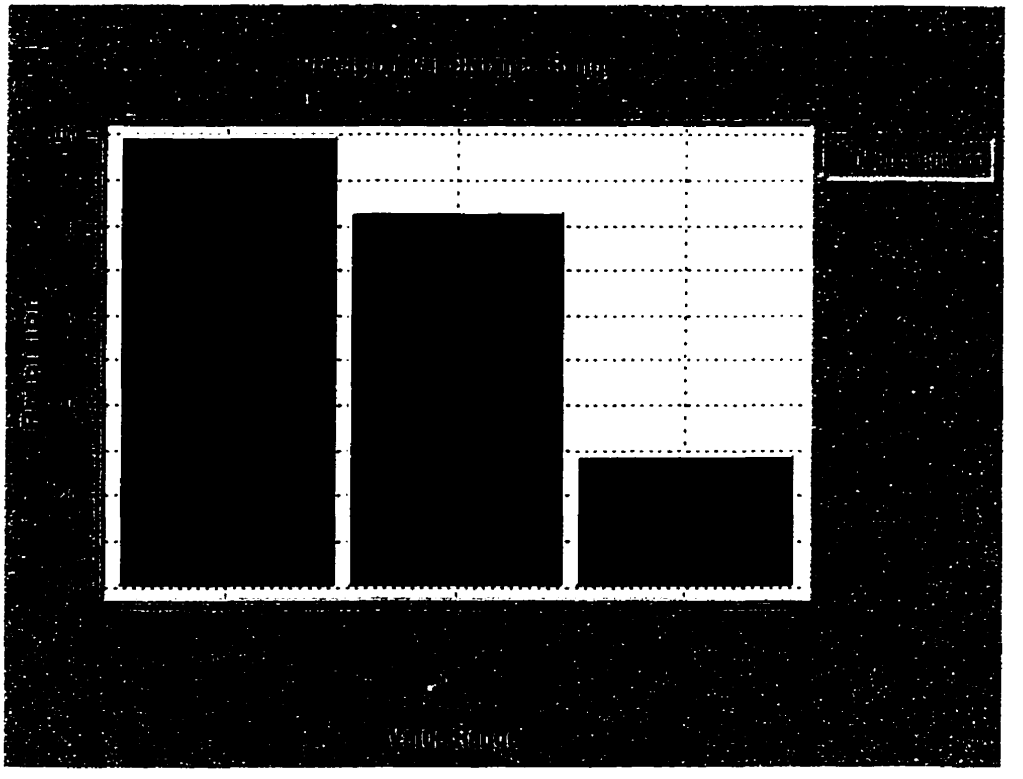


FIGURE 6.8. Average Precision (in percentage points) Per Distance Range for the Metrics criterion.

on the Precision of this method. The relation between Recall and Precision per distance range is illustrated in Table.6.7. These results indicate that with the Metric criterion in DP-based matching we have significant loss in Precision and little gain in Recall when we deviate from zero distance threshold [2]. Therefore, this method is not the best to apply, but it is a compromise between the efficiency and speed of computing and comparing metrics, as well as the increased accuracy provided by the DP-based matching. This method is to be applied when exact clones are sought, and the recognition speed is a significant factor.

[2]Zero distance threshold means that we retrieved components with distance equal to zero

Distance Range	*Precision Drop (%)*	*Recall Increase (%)*	*Total Recall Achieved (%)*
0.01 - 0.99	-16.0%	+3.0%	44.4%
1.0 - 1.49	-54.0%	0.0%	44.4%

TABLE 6.7. Recall / Precision Relation Per Distance Value Range (DP Metrics Criterion)

3.4. Precision Per Distance Range (Data Types Matching Criterion). This experiment was performed to evaluate the overall Precision behavior of the DP matching using the *Data-Types* criterion (Fig.6.9). This criterion performed very well for distances close to zero but results in a drastic drop of Precision for distances higher than zero. The reason for this behavior is that data types of variables updated or used impose only limited constraints on the structure and the quantitative values of the features that may be used to characterize and classify a software component. The fact that two statements use a variable of the same data type does not impose any other requirement on the class of the two statements compared (e.g. a While-Statement, an If-Statement) or on their detailed features (e.g. what globals are set or used in the statement, how many I/O operations are performed, how many functions are called from a statement). The Recall / Precision relation per distance range values is illustrated in Table.6.8. These results suggest that for distances close to 0.0 the Precision of this method is dropping very fast and we have not observed any difference in Recall. This can be explained by the fact that the data types criterion is not a very good one on localizing exact clones, and allows for noise to be added easily. However, this approach performed well on the Recall/Precision experiments, especially at high Recall values where Precision level stabilized at acceptable levels compared to the other criteria (please refer to Fig.6.10). The reason is that when we compare code fragments using the Data Types criterion we can obtain most of the clones just by considering the fact that the two statements use some complex Data Type. We believe that a possible explanation for the relatively high Precision obtained in this experiment is not because the approach is better than the others but because code fragments that may use a particular specialized (i.e. complex) data type are few in a software system compared to all other data types in the same system. Therefore, even at high Recall values we were able to achieve a good Precision rate. This approach is recommended when we are looking for clones where we know that their main characteristic is that they contain a very specialized data type that exists only in a few parts of a large system.

3.5. Recall / Precision Per Matching Feature Used. Precision values for specific Recall values were again computed by applying the same set of queries we have used for the Metrics-based matching approach. Average Recall values and average precision values have been used to produce with linear interpolation precision values for standard recall values (0.0, 0.1, 0.2, .. 1.0).

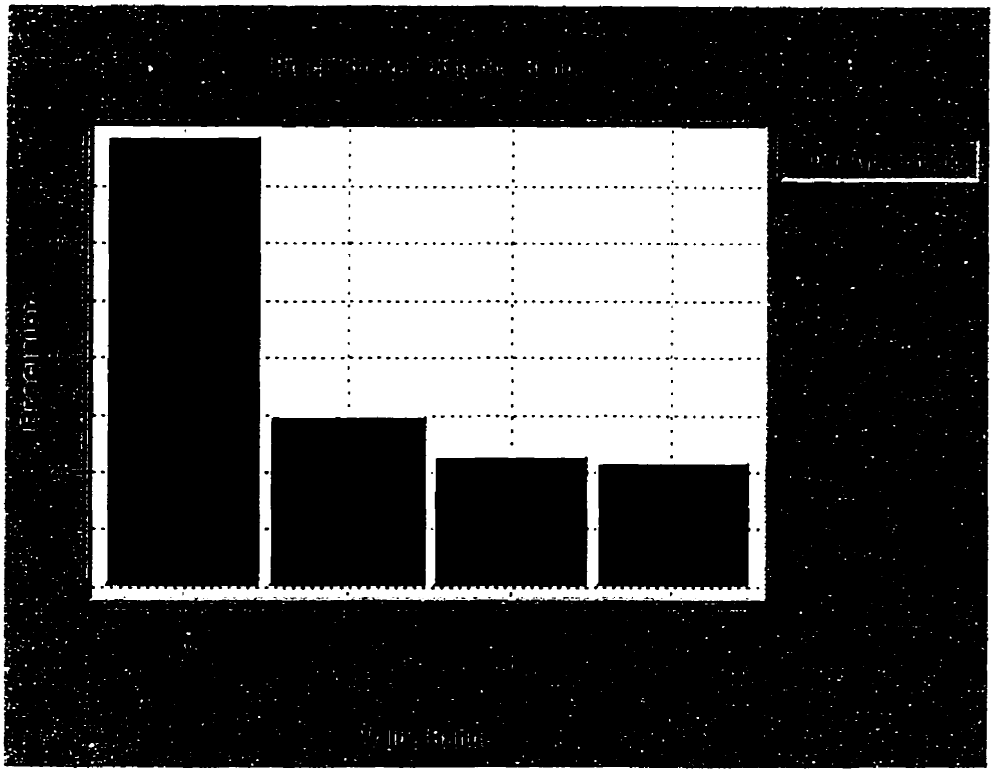


FIGURE 6.9. Average Precision (in percentage points) Per Distance Range for the Data-Types criterion.

Because of the erratic nature of low recall values for small samples [**Jones81**] we assigned a Precision value of 1.0 to Recall value of 0.0.

The obtained results indicate that:

- Using the *Set-Uses* Criterion: we have obtained the best performance on average compared to the other two approaches. Precision levels for low Recall values were in between the Precision levels achieved using the Metrics criterion, and the Data Types criterion. This behavior is due to the noise introduced in the Metrics criterion by linear interpolation as

Distance Range	*Precision Drop (%)*	*Recall Increase (%)*	*Total Recall Achieved (%)*
0.01 - 0.99	-48.0%	0.0%	17.9%
1.0 - 2.0	-8.0%	+12.1%	30.0%
2.01 - 5.99	-8.0%	+17.3%	47.3%

TABLE 6.8. Recall / Precision Relation Per Distance Value Range (DP Data Types Criterion)

Standard Recall	*Precision (Set-Uses)*	*Precision (Metrics)*	*Precision (Types)*
0.0	100.0%	100.0%	100.0%
10%	83.3%	89.2%	77.5%
20.0%	71.9%	78.4%	62.5%
30.0%	74.0%	67.6%	74.9%
40.0%	75.5%	57.6%	72.9%
50.0%	66.0%	59.7%	65.9%
60.0%	52.9%	61.6%	58.1%
70.0%	44.7%	45.8%	50.3%
80.0%	36.0%	30.9%	42.6%
90.0%	26.1%	16.8%	25.5%

TABLE 6.9. Recall / Precision Table (DP)

we were not able to achieve Recall values less than 34.4%. Therefore for low Recall values it seems that the Set-Uses criterion does not perform better than the Metrics criterion, but we believe this is mainly due to the noise introduced at the metrics Recall/Precision curve. However, the two curves are quite close for low Recall levels. At higher Recall values, the approach performed marginally worse than the Data Types criterion. The reason for this is that some of the queries contained data types that could be found only in parts of the system while the use of Global Identifiers (that introduced noise under the Set-Uses criterion) were evident in large parts of the system, and therefore allowed more candidate components to be considered. However, the Set-Uses approach is safer to use in general queries as it does not impose as strict constraints on the parts of the system that can be retrieved.

- Using the *Metrics* Criterion: we have obtained better performance than using the metrics-based matching (simple metrics comparison), but not as good as the overall performance obtained by the Set-Uses and Data-Types criterion. The reason for this result is that the Metrics criterion looses its relevance at such a low level comparison granularity (i.e. statement level). The increased performance at low Recall values is reported due to the linear interpolation used for Recall values less than 34.4%. This approach is a good compromise between the speech of the metrics-based matching and the added accuracy of the DP-based

Average Code Size	30,392 LOC
Min Segmentation Time	3 secs.
Max Segmentation Time	184 secs.
Average Segmentation Time / Query	38.06 secs.
Min Matching Time	32 secs.
Max Matching Time	3619 secs.
Average Matching Time / Query	638.74 secs.

TABLE 6.10. Performance Statistics for 100 queries in three software systems (Tcsh, Clips, Roger)

matching. This approach is to be used when large software packages are analyzed and the programmer would like to balance the speed and accuracy of the obtained results.

- Using the *Data Type* Criterion: we have obtained marginally the highest performance among all the other approaches. The reason for this result is that our queries contained Data Types that are found only in a few parts of the system and therefore the retrieved items were fewer than the number of items retrieved by the other approaches. This phenomenon forced the Precision level to be slightly higher than the others. Therefore, this criterion is promising and can be used when the pattern sought contains specialized Data Types that are used only in parts of a system. However, it is not safe to assume that all queries a user asks may fall in this category, and the slight gain in Precision compared to the *Set-Uses* criterion is not a significant factor for choosing the *Data Types* criterion which imposes constraints (i.e. good performance when complex data types are present).

4. Markov-based Matching Experiments

4.1. Performance Statistics. In Table.6.10 performance statistics for the Markov-based approach are illustrated. This experiment was conducted by formulating 100 ACL queries and performing pattern localization in *Clips*, *Tcsh* and *Roger*. One observation is that the code delineation criterion is quite efficient in localizing candidate code fragments to be considered by the pattern matcher. The average delineation time is 38.06 seconds which is an acceptable performance given the average code size. Another observation is the rather high average matching time reported in these queries. One reason for this is that most of the queries contained several *wild character* statements, and utilized the *interleaving* and the *choice* operator. Furthermore, performance was degraded due to the frequent garbage collection operations utilized by the LISP environment in which we implemented the matcher. Undoubtedly an implementation of the matcher in a faster programming language (i.e. C) may provide a match better performance. However, the LISP implementation in

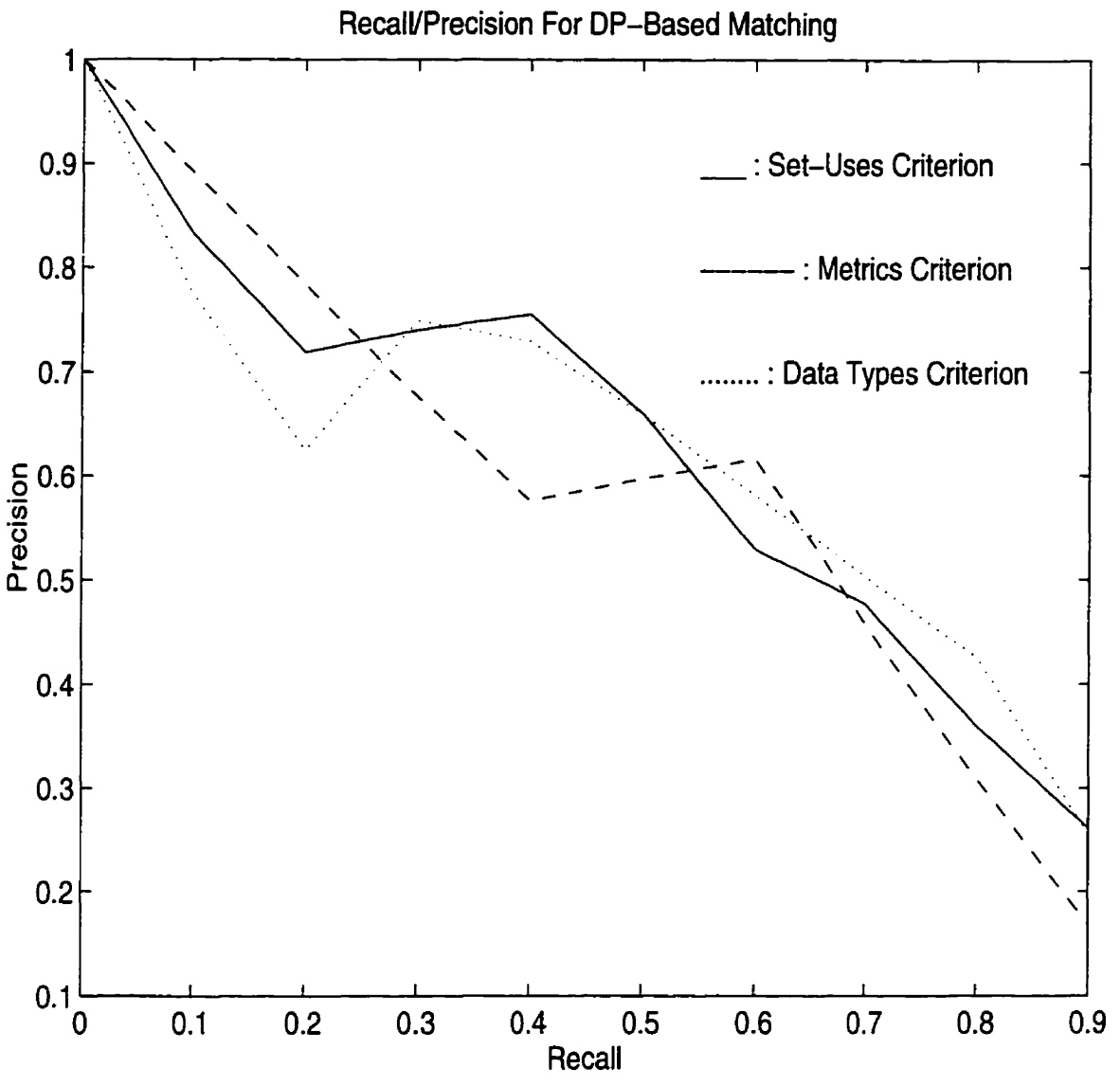


FIGURE 6.10. Recall/Precision for DP-based Matching.

all of the discussed experiments performs better than other existing implementations [**Magdal96**], [**MacLaugh95**].

4.2. Recall / Precision Comparison. Precision values for specific Recall values were again computed by applying the same set of queries we have used for the metrics-based matching approach. Average Recall values and average precision values have been used to produce with linear interpolation precision values for standard recall values (0.0, 0.1, 0.2, .. 1.0). Because of the erratic nature of low recall values for small samples we assigned a Precision value of 1.0 to Recall value of 0.0.

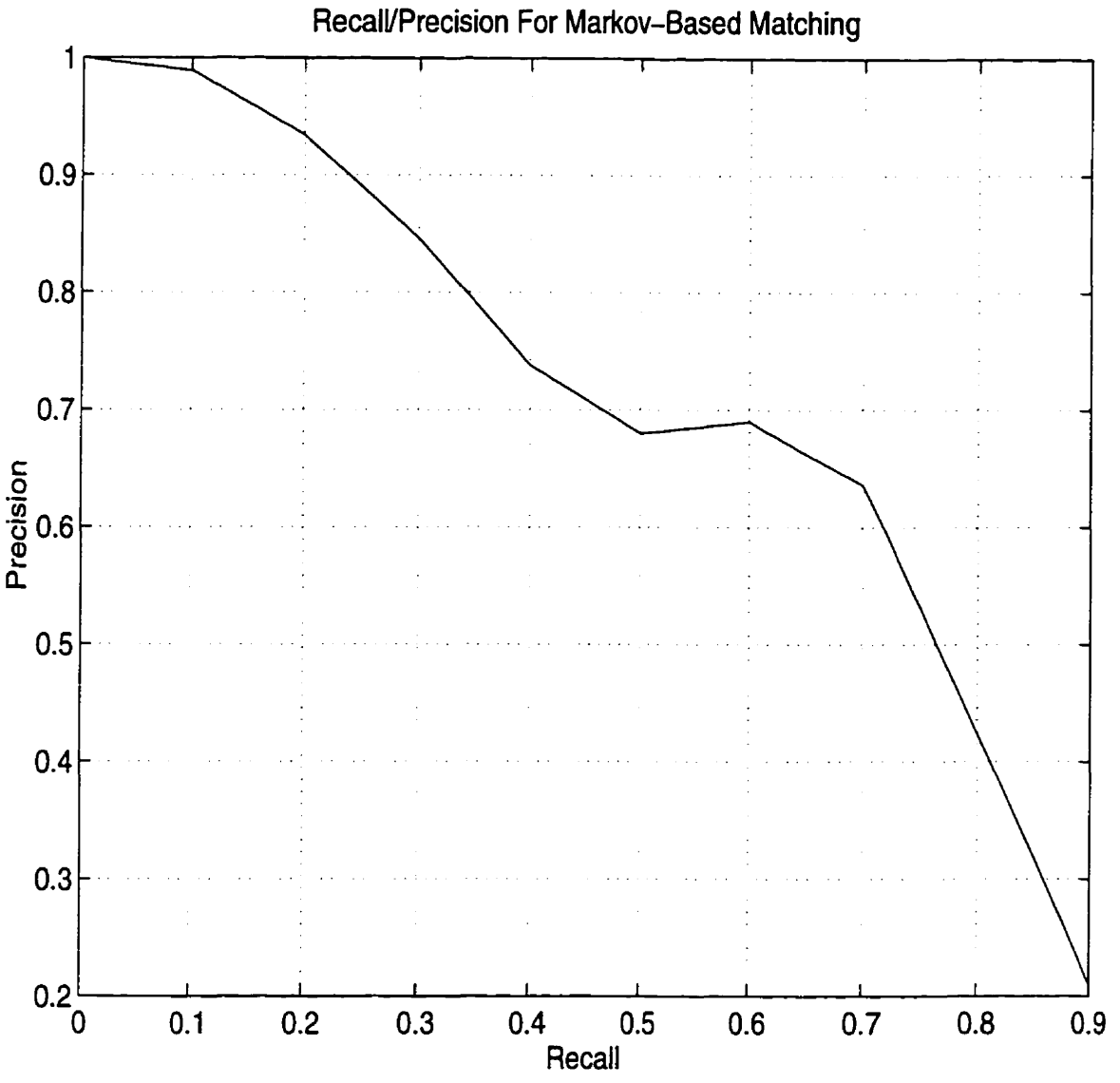


FIGURE 6.11. Recall / Precision graph for the Markov Based matching.

The results obtained using this technique were clearly the best among the approaches we have experimented with. The major observations on the Recall/Precision curve obtained are:

- the Precision levels are maintained at high values for higher Recall levels when compared with the best curves obtained for the metrics-based approach and the DP-based approach.
- the performance of this approach is the best for Recall levels approximately in the range of 40.0% - 70.0%, but is comparable with the performance of the DP approach for Recall levels lower than 50%. The reason for this behavior is that components that are structurally similar with no major modifications can be captured by both the DP-based method and

the Markov-based method. However, when modifications are introduced (i.e. changes in variable names, changes of statement types), the DP-based method may reject candidates based on threshold cut-off values, while the Markov approach will allow more matching flexibility in this cases (i.e. will succeed in matching a *Abstract-Conditional-Statement* with an *If-Statement* or a *Switch Statement* while the DP will fail and will effectively consider such a comparison as an insertion or deletion).

- The drawback of the approach is that we could not effectively achieve, on average, Recall values higher than 78.4%, so that any Precision value obtained for Recall level higher than 78.4% was computed using linear interpolation on the assumption of a Precision value 0.0 for Recall value 1.0. This is the most pessimistic assumption but ensures that the resulting curve is the lowest that can be achieved. In our experiments, we had queries that achieved a Recall level higher that 78.4%. These queries gave an overall average Precision of 30.0% which is a more realistic estimate, for the performance of this approach.

5. Overall Recall / Precision Comparison

In this section we present a comparison the Recall/Precision curves for the best combinations observed in each of the pattern matching techniques proposed. In Fig.6.12 these Recall/Precision curves are illustrated.

- Clearly, the Markov-based approach performs better that the other two. The basic advantage of this approach is that it maintains a high Precision level for high Recall levels. With our experiments we effectively achieved an average Recall value of 74.8% for which we we obtained, using the Markov-based approach a Precision of approximately 40.3%. The linear drop in Precision we observe for Recall values higher than 74.8% is a product of linear interpolation taking the most pessimistic assumption that at Recall level 100.0% we have Precision 0.0%. In practice this assumption does not hold. In fact those of our experiments that achieved effective Recall values higher than 74.8% suggested a Precision value of 30.0%. The drawback of this method is that it is slower than the other two, especially when the formulated queries involve the *choice* and the *interleaving* operator. We think that significant performance enhancements can be achieved by implementing the algorithms in a more efficient programming environment. This approach is suitable for cases where high Precision and high Recall values are sought.
- The DP-based method is the second best approach of the ones we have tried. For Recall levels lower than 45.0% the algorithm seems not to perform as well as the metric-based one. This behavior can be explained by the interpolation error introduced at low Recall values

of the metrics-based approach. Note that in the metrics-based method, the lower effectively achieved Recall value was 44.3% at distance 0.0. Recall values from 0.0% to 44.3% were interpolated. This explains the linear drop in Precision for Recall values less that 44.3%. The overall acceptable Recall/Precision performance of this approach combined with its good time performance make it a very attractive candidate for cases where the user wants to balance between accuracy and speed. Clearly this approach is very suitable for large systems in which exact clones are sought.

- The metrics-based method performed very well for Recall values less that 44.3%. For higher Recall values we observed a linear drop in Precision up to Recall values of 82.5%. For Recall values higher than 82.5% the method seems to stabilize at a Precision level of 10.2%, which is lower than the Precision levels achieved using the other two methods at this Recall level. We can safely conclude that this method trades speed for accuracy in the obtained results. The metrics-based method is very suitable for large systems in cases for which only part of the clones are sought. Moreover, this method can be applied as a preprocessing step to limit the search space of a more accurate but slower matching technique such as the DP-based or the Markov-based ones.

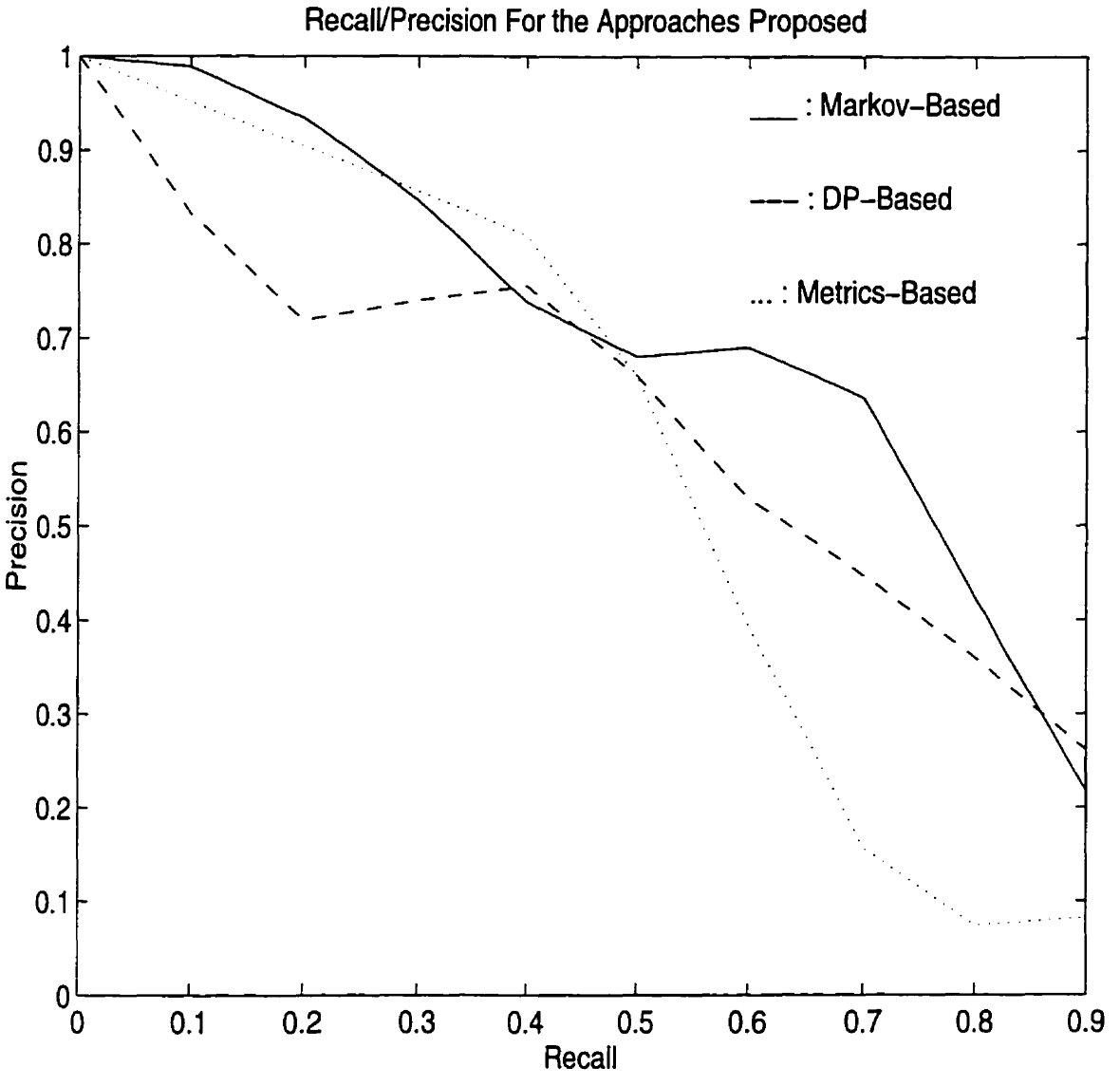


FIGURE 6.12. Recall / Precision graphs for the pattern matching methods proposed.

CHAPTER 7

The System Architecture

1. Communication with other Tools

Design recovery is a process that produces high-level descriptions of a software system from an analysis of its implementation. Usually this means extracting requirement or specification concepts from a program's source code. Once design decisions are recognized, they are then represented in a form that is suitable for subsequent use. This process requires the integration of different technologies, perhaps in more than one software environment, and with a variety of analysis tools. In developing an integrated design recovery environment, the following important problems have to be considered:

(i) Program representation,
(ii) Adaptation of data, control, and presentation to the components of the environment,
(iii) Extension of the data models and interfaces to support the registration of new tools and user-defined objects, dependencies, and functions,
(iv) Development of domain-specific, syntactic and semantic pattern matching for plan localization and recognition,
(v) Representation and support of processes and methodologies for reverse engineering,
(vi) Construction of robust user interfaces, and algorithms, capable of handling large collections of software artifacts.

Data integration is essential to ensure data exchange between tools. This is accomplished through a common schema.

Control integration enhances inter-operability and data integrity among different tools. This has been accomplished by the development of a server that handles requests and responses between tools in the environment [**Wasser90**].

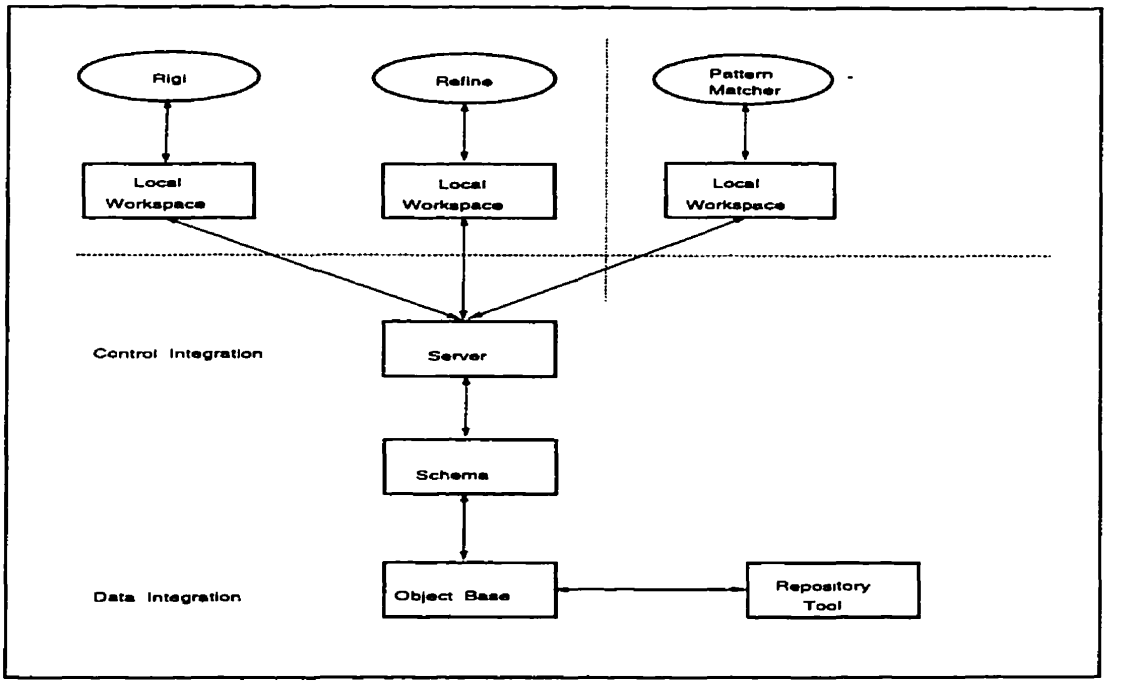


FIGURE 7.1. The implemented system architecture for tool integration. Dashed lines distinguish computing environments, usually running on different machines.

A solution to the above mentioned problems is based on a system architecture in which all tools communicate through a central software repository that stores, normalizes and makes available the results of analysis, the design concepts, the links to graph structures, and other control and message passing mechanisms needed for the communication and cooperative execution of different tools. Such integration is achieved by allowing a local workspace for each individual tool in which specific results and artifacts are stored, and a translation program for transforming tool-specific software entities into a common and compatible form for all environments entities.

The translation program generates appropriate images in the central repository of objects shared with local workspaces.

A system architecture for such an integrated reverse engineering environment is illustrated in Figure 7.1.

Communication in this distributed environment is achieved by scripts understood by each tool using a shared schema data model for representing data from the individual local workspaces of each tool and, a message passing mechanism for each tool to request and respond to services.

A customizable and extensible message server named *Telos Message Bus (TMB)* [**Mylo96**] handles data communication. This message server allows all tools to communicate both with the

repository and with each other, using the common schema. These messages form the basis for all communication in the system.

The central repository is responsible for *normalizing* these representations, making them available to other tools, and linking them with the relevant software artifacts already stored in the repository (e.g., the corresponding nodes in the abstract syntax tree).

Communication scripts are in the form of s-expressions. Each s-expression is wrapped in packets called *network objects* and sent via *TMB* to the appropriate target machine. *TMB* uses Unix sockets for the communication and utilizes the network infrastructure.

An example s-expression for a function called *hostnames_matching* is given below. The same function is illustrated as an object in Fig.7.2 where the repository browser provides a graphical view of the repository contents.

```
(Function_176 Token
   (Function)
   ()
   (((refineName)
     (("#176")))
    ((fanout)
     ((10.0)))
    ((cyclomaticComplexity)
     ((5.0)))
    ((functionPoint)
     ((115.0)))
    ((calls)
     ((Function_175)
      (Function_174)))
   ((definesLocals)
     (("word")
      ("number")
      ("need_here_doc")
      ("yylval")))
   ((definesGlobals)
     (("shell_input_line")
      ("pushed_string_list")
      ("shell_input_line_index")))
```

```
((functionName)
 (("hostnames_matching")))))
```

The shared schema facilitates data integration and encodes program artifacts (i.e. the AST), as well as analysis results (e.g. the call graph, the metrics analysis).

A system integrating Refine, Rigi and Telos running simultaneously at three separate sites (Toronto, Victoria and Montreal) has been implemented and is currently in use.

This architecture has been further expanded to accommodate control and data integration between different tools and processes in a Cooperative Information System Environment [**Mylo96**].

1.1. Data Integration. Data Integration has two major objectives, namely:

- to represent the source code in the centralized repository and make representations available to other tools even if they use a different program representation scheme in their local repositories,
- to model the analysis results in order to make them available to other tools for subsequent analysis.

The Data Integration schema consists of two parts. The first part captures information about the program syntactic structure currently used by Refine and Rigi. Multiple inheritance allows encoding of information that may not be stored in all local workspaces.

The schema is modeled in Telos [**Mylo90**] and realized in ObjectStore, a commercial Object Oriented Repository. A part of which is illustrated in Fig.7.2.

The second part classifies the patterns used, and combines the analysis results generated by different tools.

1.1.1. *Schema for Program Representation.* We have selected the AST as the basis for program representation in the centralized repository. The choice is motivated by the fact that the AST is a direct product of the parsing process, can be created by any parser for the target language in various degrees of detail and granularity (i.e. detailed parsing versus light-weight parsing), and many other program representation schemes (e.g. program dependency graphs, combined data/control flow graphs) can be considered as analysis (result) and annotations of the basic AST. Results obtained with different tools can be linked via annotations to the AST nodes.

The schema has been implemented in Telos [**Mylo90**] and allows for multiple inheritance and specialization of the attribute categories for each object. In this way, an AST entity (i.e. File) may have attributes that are classified as *Refine-Attributes* or *Rigi-Attributes* and be encapsulated in the

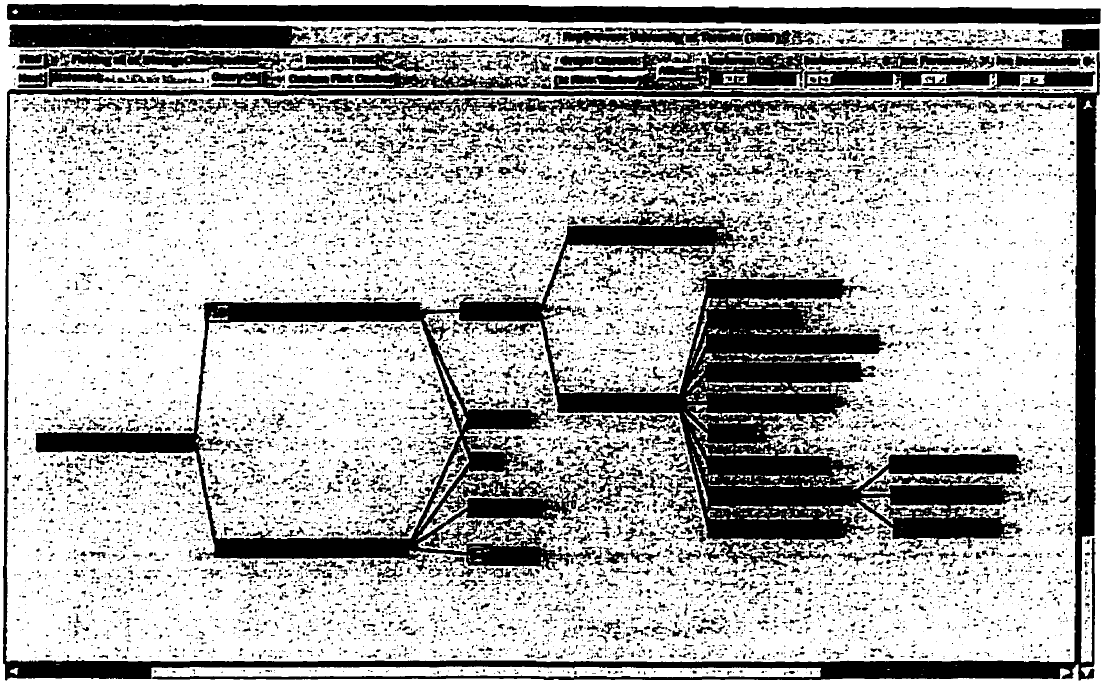


FIGURE 7.2. Part of the Schema hierarchy. Multiple inheritance is shown for the *File* and *Module* nodes.

same schema object. The Schema is populated by tools that may add, or retrieve values for the attributes in the objects to which it has access to.

At run-time, the user or the communication scripts may request and select only the attributes, and their corresponding values, that are relevant to a particular tool. The server has been extended to handle more complex messages and respond automatically to events using a rule base and the *Event, Condition, Action* paradigm [**Mylo96**].

An example of the schema structure for the *File* AST node as it is represented in the central repository is illustrated in Fig.7.3

1.1.2. *Schema for Analysis Results.* The basic purpose for tool integration is the development of an environment in which each tool exchanges analysis results with other tools so that the overall functionality of the system will be enhanced and be bound to the functionality of each individual tool as if it were stand alone.

Data integration for analysis results was achieved by designing Schema classes for every type of analysis a tool is able to perform. Each such class is linked via attributes to the actual AST entities and this is visible to all the other tools that may request it.

For example, in Ariadne, a *Cluster* is a collection of functions and can be generated by a specific type of analysis (e.g., data bindings related clustering, similarity based clustering) while in Rigi this

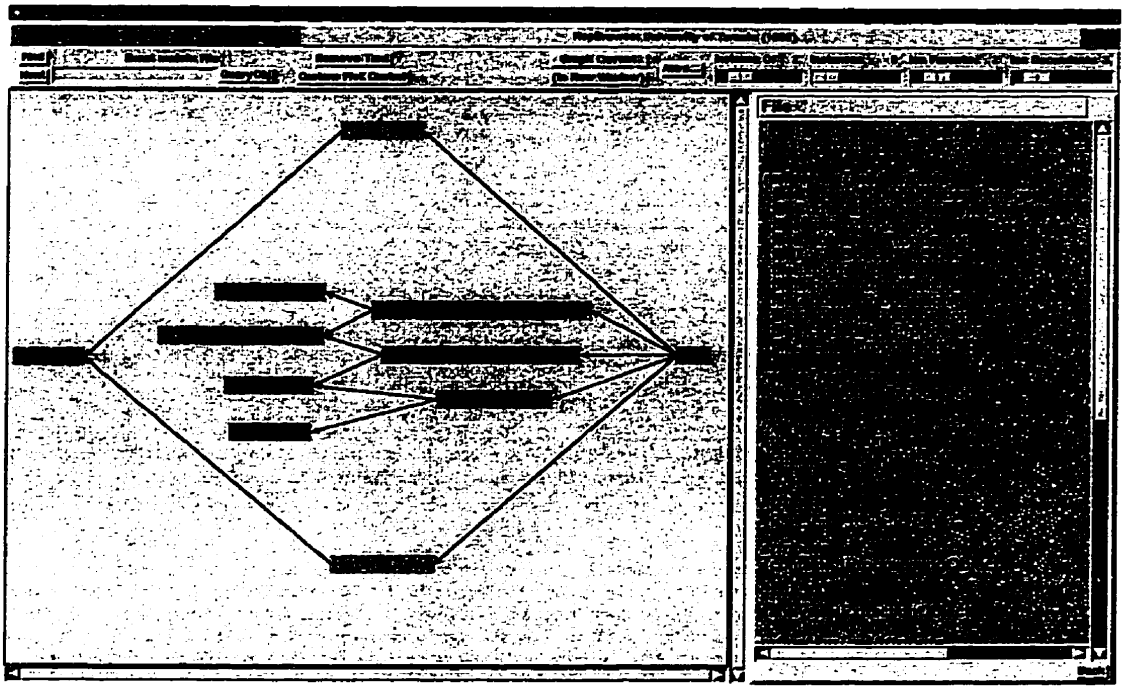


FIGURE 7.3. The Schema structure and inheritance for the *File* AST entity. The Refine-Attributes and the Rigi-Attributes are encapsulated in the same object in the central repository.

is considered a *Subsystem*. In the central repository, both are encoded using the Module Schema class which is used thereafter for communicating analysis results involving grouping of software artifacts and system partitioning. A reference table is used in the central repository to map and normalize class names from each individual tool. This table has been built a-priori as the user registers the services that each tool offers manually when designing the system. New services can be added by updating the reference table, and adding the appropriate new Schema classes.

The Ariadne Schema related to the services Ariadne offers is illustrated in Fig.7.4. Focus has been placed on the *RegularExpressionLocalization* service which localizes code given a regular-like description of its structure. Note that the result can be made available to the other tools by the attribute *regExpressionLocalizationRes* which can take as values instances of the *Module* schema class. When such an analysis is performed, a new Token of type *RegularExpressionLocalization* is created and the attribute *regExpressionLocalizationRes* contains a collection of software artifacts that match the given regular expression which is stored in the *regExpression* attribute. When Rigi or another tool requests the results of this type of analysis, it gets as a response the value of the *regExpressionLocalizationRes* attribute which is a set of software artifacts or, more precisely, a set of pointers to the corresponding object nodes that represent these software artifacts in the AST. Once

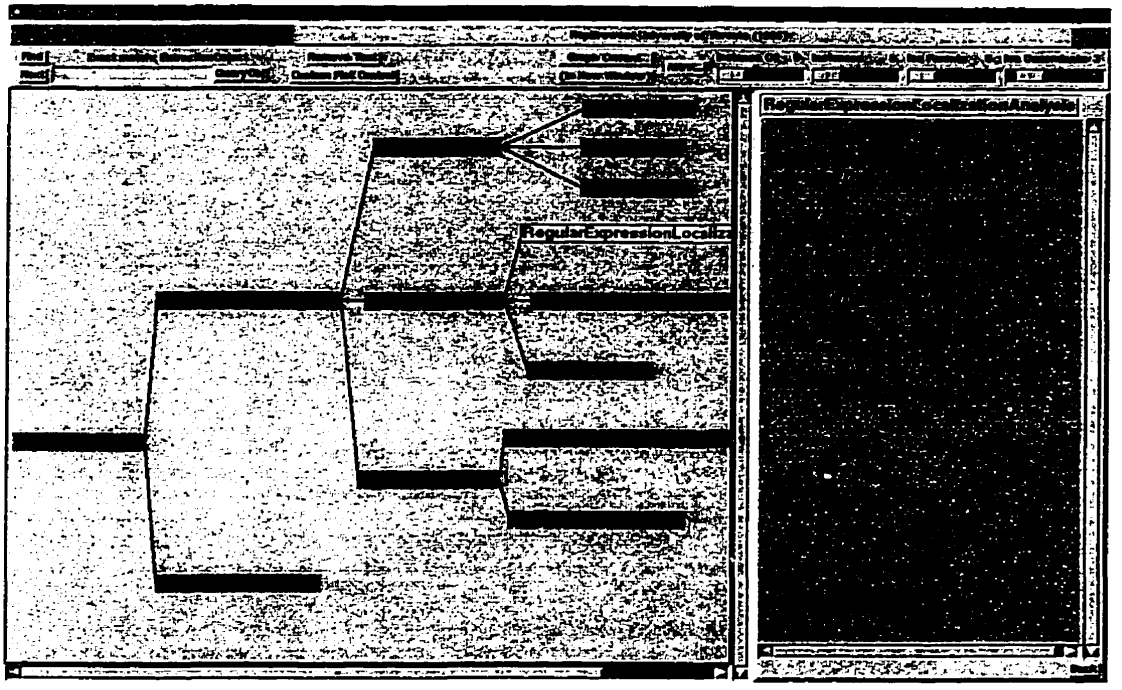


FIGURE 7.4. ExtractionObject Schema hierarchy for the Ariadne System.

pointers to the AST have been passed, all other types of analysis available in Rigi can be performed locally and applied to this specific *Module*.

1.2. Control Integration. Control Integration is based on designing and developing a mechanism for :

- Uniquely registering tool sessions and corresponding services,
- Representing requests and responses,
- Transferring object entities to and from the repository,
- Performing error recovery.

At the Application layer [**Stallings91**] *Mbus* is used to facilitate inter-networking. *Mbus* offers an environment to manage the transfer of data via TCP/IP using a higher-level language to represent source and destination points, processes, and data. *Mbus* offers an environment to access lower level UNIX communication primitives (i.e. sockets), in order to manage the transfer of data via TCP/IP using a higher-level language to represent source and destination points, processes, and data.

Each tool generates a stream of *network objects* encoded as a stream of s-expressions. A parser analyzes the contents of each *network object* and performs the appropriate actions (e.g. respond to a request, acknowledge the successful reception of a *network object*). Moreover, a mediator program is

used to check if a request or an acknowledgment has arrived, and initiates the appropriate actions. In particular, UNIX named pipes are used to signal an incoming event and handle incoming and outgoing data. The mediator program constantly checks for the existence and the status of these pipes and initiates the appropriate processes for reading or writing.

The system handles the following messages:

- Ask for instances of a particular object class from a local workspace or the central repository (ASK type messages),
- Add new instances of an object class to a local workspace or to the central repository (ASK type messages),
- Add new values to attributes of a particular object (TELL type messages),
- Ask for the status of a tool (i.e. running, registered) (ASK type messages),
- Acknowledge the successful termination of a request (ACK type messages).

Messages can be send from each tool to any other tool

- Via the centralized repository or
- In a direct fashion.

The normal mode is for the tools to communicate via the central repository; in this mode each tool contributes a specific analysis that another tool can use and enhance. The whole process is driven by the user who is aware of the types of analysis he or she has performed or are already performed by other users and stored from previous sessions.

The direct communication mode has been implemented to facilitate development and is not intended to be used as the standard communication mode.

Messages are classified in two categories:

- Point-to-Point messages
- Broadcast messages

Point-to-Point messages have unique origin and destination points. A message is uniquely represented by:

- The source (i.e. the tool that initiated the message),
- The destination tool,
- The time that the message was issued,
- The type of the message (ASK, TELL, ACK).

An origin or a destination point (i.e. a specific session of a Rigi tool) is represented by:

(i) Its process Id,

(ii) The user who owns the process,

(iii) The host name,

(iv) The time the process started,

(v) The name of the tool.

A typical example of *Point-To-Point* communication is the request from a particular Ariadne session for analysis performed by a specific Rigi session (i.e. a Rigi process).

Broadcast messages have a unique source and multiple destinations. A typical example of a Broadcast message is the request from a particular Ariadne session for analysis performed by all Rigi sessions that are registered or have been registered in the past in the environment.

Each tool session is represented in the central repository as an instance (Token) of a particular object class (i.e. AriadneSession, RigiSession). Checking for the status of a tool session is achieved by querying the central repository.

Acknowledgment of the successful termination of a request is performed by issuing a point-to-point message that contains:

(i) The identification of the original message,

(ii) The identification of the tool that received the message and successfully completed it

Currently, the repository operates in a *monotonic* mode, where changes of the attribute values are not allowed as this would have required the development of a Truth Maintenance System to maintain logical consistency in the central repository, as well as in the local workspaces.

In future versions of the server new functionality can be added in order to handle more complex types of messages (i.e. RETELL) and allow for changing and maintaining attribute values.

1.3. Integration Statistics. In this section the integration statistics are discussed and in particular the relationship between source code size, total number of repository generated objects, data retrieval performance as well as upload and down-load times.

The experiments involved four software systems, *Twentyone*, *bash*, *tcsh* and, *Clips*. For each system we have measured:

- the total number of objects that represent parts of the AST in the central repository,
- down-load performance by measuring
  - the total number of objects in KB,
  - the total number of objects retrieved by selection queries and,
  - the Dow-load time per query,

System	*LOC*	*# of Functions*	*# of Files*	*# of AST Repository Objects*
TCSH	44,754	658	46	3,340
CLIPS	32,807	705	40	1,694
BASH	27,393	632	63	1,606
ROGER	13,615	235	39	1,089
TWENTYONE	943	38	3	920

TABLE 7.1. Storage Statistics (only File and Function object types stored)

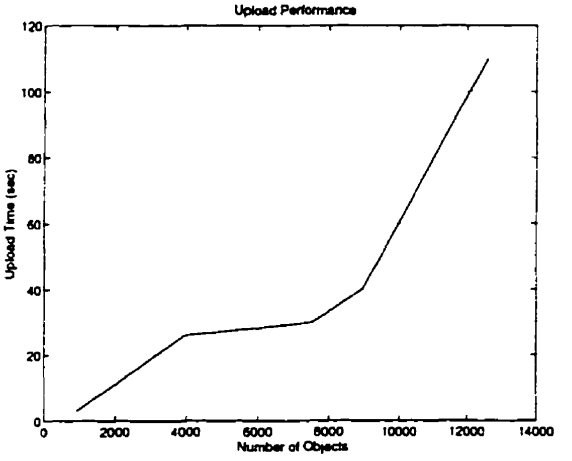


FIGURE 7.5. Upload Performance

- upload performance by measuring
  - the total number of objects to be loaded to the repository,
  - the upload time.

The total number of objects generated for the reduced AST for the four subject systems is illustrated in Table.7.1. These measurements indicate that the approach of storing only the necessary parts of the AST results in a large potential for scalability, as major increases in the size of the source code do not affect dramatically the total number of objects generated.

The upload times are illustrated in Fig.7.5. These statistics indicate a relatively linear relation between the upload time and the total number of objects to be loaded in the repository. The total number of objects in this experiment was obtained by allowing objects that represent statements to be generated as well.

Similarly, the down-load statistics are shown in Table.7.2. These statistics indicate that down-load time depends on the size of the objects retrieved and not on the size of the repository. This is an important observation as it is directly connected with the scalability of the system.

System	*Total Objects in KB*	*Objects Retrieved by Query*	*Down Load Time (sec)*
TCSH	3340	658	10
TCSH	3340	47	1
CLIPS	1694	705	15
CLIPS	1694	41	1
BASH	1606	632	10
BASH	1606	63	2
TWENTYONE	117	38	1
TWENTYONE	117	3	1

TABLE 7.2. Dow-load Performance (KB contains File, Function type objects)

CHAPTER 8

Conclusion

Many of the problems related to Software Maintenance originate from the overall poor condition of large systems in terms of complex source code and obsolete documentation. The essence of Software Maintenance problem, though, may be ultimately traced back to the lack of sufficient understanding of the structure, functionality, characteristics and component dependencies in large software systems.

This thesis has reviewed the state-of-the-art for Program Understanding techniques and discussed how these approaches address problems related to software maintenance. It is evident that the research community in Program Understanding proposed a lot of interesting ideas that originate from different areas. Two of the key themes in Program Understanding research, as it has evolved the last four years, are *plan recognition* and *data flow analysis*.

Within this framework, some teams have chosen to apply compiler techniques to compute and analyze Program Dependency Graphs, perform slicing, value range analysis, constant propagation analysis, and symbolic evaluation. These techniques have been used to identify parts of code that may be relevant to a particular maintenance task.

Other teams have used customized graph-based program representation formalisms to represent code abstractions and proposed techniques to match these abstractions against the contents of a static repository that contains representations of programming plans. These techniques have been used to identify commonly used algorithms as well as application domain specific programming plans in a software system.

Approaches to plan recognition must address two key questions:

(i) Whether a plan based approach can be effective as it implies a plan library that must contain all possible plans in advance,

(ii) How program understanding algorithms can scale-up and be applied to large software systems as most existing program understanding algorithms that rely of flow-graph matching are NP-complete in the worst case.

We have chosen to work on problems related to Plan Recognition and in particular, the clone detection, and the concept assignment problems.

Code cloning which is a widespread practice among developers;

a) increases the complexity of a software system, and the size of the corresponding executable program.

b) increases maintenance costs as changes in one component have to be propagated to all of its cloned instances

c) code cloned components are prime candidates for repackaging and generalization to a software repository of reusable components

The concept assignment problem consists of attaching a "meaning" to a code fragment, by understanding its overall functionality, data and control flow properties, as well as its possible abstractions.

Within this framework our hope was to design scalable pattern matching techniques that can be used to locate and retrieve instances of code cloned components in a large software system and provide a formalism to represent (in a higher level of abstraction), implementations of generic, as well as, domain specific, algorithms.

The thesis proposes three matching techniques that are scalable, do not depend on a plan library, and are flexible and modifiable by the end user.

1. Contributions

This section summarizes the major original contribution of this thesis.

Firstly, we have proposed and experimented with a number of program features that are used to compute five standard software engineering metrics that classify and represent a code fragment. We have shown that these metrics can be computed compositionally by using the Abstract Syntax Tree at link time. This matching technique is based on the assumption that if two code fragments are clones, then they share a number of structural and data flow characteristics that can be effectively classified by these metrics. We have shown that the metrics-based approach provides a fast approximation of the code cloning recognition problem. Experimental results have indicated that we can effectively retrieve 60% of the code cloning instances sought, and maintain a Precision of approximately 41.0% at the final results. The strength of this approach is that it can be easily used, does not depend on any complex formalism to represent source code entities, and it is time and space efficient, as it is

mostly based on comparison of numeric tuples. The price to pay for the speed and ease of use in this method is that at higher Recall levels noise can be introduced and low Precision values be obtained. At a Recall level of 70.0% the Precision can drop to 19.2%. However, this is not problematic as only a small fraction of the system is retrieved (in our experiments $\preceq$ 11.3% of the total size of the system) and therefore can be used at a pre-processing stage to limit the search space when using more accurate but more computationally expensive methods.

Secondly, we have designed and implemented a Dynamic-Programming algorithm that uses data and control flow information to compare two code fragments and evaluate an overall dissimilarity measure based on insertions, deletions and substitutions of basic statements and expressions that occur in the two code fragments compared. Dynamic Programming algorithms are very fast and efficient and have been extensively used in real-time applications (e.g. speech recognition systems). The Dynamic-Programming approach is shown to be more accurate than the pure metrics-based method but it is computationally more expensive. In particular, its complexity is on the order $O(n * m)$ where n is the number of statements of the first fragment considered to be the *model* and, m be the number of statements of the code fragment matched against the *model*. However, this type of complexity is still fully acceptable when considering large systems. Experimental results have shown that we can effectively retrieve 70% of the clone instances of a *model* code fragment and still maintain a Precision of approximately 45.0%. Moreover, this method performed very well on even higher Recall values where, for Recall levels of 90%, we maintained a Precision level of 26.1%.

Thirdly, we have proposed and implemented a concept description language ACL that is used to represent programming patterns. ACL is used to represent data flow, control flow, and data type properties of programming patterns. A matching mechanism that is based on Markov Models is used to establish correspondences between a parse tree of the concept description language and the source code. The Viterbi algorithm is used to identify the best alignment between a pattern represented in ACL and a code fragment. Experimental results obtained by examining medium sized software systems, have illustrated that this approach is powerful enough to localize programming patterns more effectively than the pure Dynamic-Programming or the Metrics-based methods. Moreover, the complexity of this method is again on the order $O(n * m)$ where n is the number of Abstract Statements in an ACL pattern that is used as *model*, and m is the number of statements in a code fragment matched against the *model*. Our experiments have effectively achieved an overall average Recall level of 70.0%, maintaining an overall Precision of 63.6%. Interpolated values for 90.0% Recall levels have suggested pessimistic Precision levels of 22.0%. However, those of our experiments that have achieved effective Recall levels of close to 90% have suggested Precision close to 30.0%, which is a more realistic estimate.

This method is not restricted by the particular program features used for matching and, in a way, ACL can be considered as a vehicle that allows for new program features to be added and considered during the comparison process. Moreover, we have shown that this technique is scalable and hierarchical abstractions of programming patterns can be effectively built from descriptions of a number of simpler base patterns.

The proposed technique can be applied to localize patterns in a variety of Procedural languages such as PLI, PL/X, PL/AS, Pascal and C.

These techniques have been implemented to provide a Program Understanding tool-set that has been integrated with a visualization tool and a software Repository. The system runs in a distributed environment utilizing Unix sockets and the TCP/IP protocol, allowing for multiple software development and maintenance teams to access and analyze a large software system.

2. Discussion and Future Work

In many ways the results reported are very encouraging. Most of our experiments have illustrated the scalability and the effectiveness of the methods proposed. Yet, in order to assess and evaluate the significance of our results it is important to discuss generic issues related to code cloning, plan recognition and the concept assignment problem.

Program Understanding has for a long time, been seen as a pattern matching problem, where programming plans represented in some formalism are stored in a static plan-base and are matched against compatible representations of the source code stored in a code-base. The way source code is represented and used by the localization and matching component of such a Plan Recognition system is fundamental for the performance and the accuracy of the results obtained.

We feel that there is an important trade-off between tractability and accuracy. Formal methods have not been very successful in recognizing Programming Plans in programs larger that 1000 lines [**Quilici96**]. However, they provide a solid foundation for Software Engineering research as they allow for semantic abstraction and effective partial recognition that may handle structural variations, implementation variations and idiosyncratic code.

On the other hand, methods that are based on measurements and analysis obtained by examining solely the Data and Control Flow properties of a subject system, fail to provide insights into the semantics of the source code examined. However, these approaches scale up and have been successfully applied to recognize Programming Plans and code duplication in large industrial systems where formal methods have failed to produce acceptable results.

In this thesis we have tried to provide generic mechanisms to bridge this gap between the pure semantic and the pure structural text-based approach to representing and localizing program patterns.

With the metrics-based approach we have tried to abstract a number of Data and Control Flow properties of the source code and effectively represent them in a 5-d numerical space. The five metrics we used were selected on the merit of their coverage of a variety of code properties to which they are sensitive. However, this method can not effectively provide any qualitative measure about how a *model* and an *input* components differ. A possible further avenue of research for this method is to examine the use of software metrics with respect to particular contexts, and maintenance objectives. This may result in guidelines for associating specific deviations of metric values with specific modifications in the structure and the logic of a subject system, thus inferring, some semantic content of the modifications involved. This type of research is essential for establishing a qualitative characteristic on software metrics that have not been thoroughly interpreted yet.

The Dynamic-Based approach is an attempt to provide a method for measuring some qualitative characteristics of the modifications that may appear between code fragments. Still, this method depends on the structural characteristics of the code examined and gives little insight into the semantic properties of the source code. However, the DP function proposed provides abstraction mechanisms that the pure text-based and metrics-based approaches do not. Moreover, the DP-Based matching subsumes these approaches, as any successful substring comparison or metrics comparison is effectively captured by the Dynamic Programming approach as well. Moreover, fixed text substitution mappings applied for parameterized text-based matching [**Baker94**], [**Baker95**], are more restrictive than the use of abstract identifier bindings and lexicographical distances proposed within the context of this thesis.

Overall, code-to-code matching and code cloning detection have started gaining attention as potential tool for software evolution and especially for software migration from Procedural to Object Oriented languages. Specifically, similar code fragments that can be parameterized on different data types may suggest the way these data types can be abstracted to classes and how associated operations can become methods to these classes. Initial work in this field has already been reported in [**Konto96b**].

Finally, the Markov-model approach is the most flexible and extensible of all three methods discussed in this thesis. The strength of this method is that it provides a framework in which alternative Programming Patterns can be represented in higher levels of abstraction and matched against specific instances of the source code. Similarity distances can be computed based on the likelihood a programming structure is generated by a Programming Pattern. The method has

been proposed in conjunction with ACL, but we consider that other specification and program representation languages can be used instead. Within this framework two possible avenues of research can be explored. The first is to enhance ACL in order to handle more semantic content. This includes the ability to specify data flow related constraints for each Abstract Statement represented in ACL, to represent data dependencies, and access to diverse sources of information related to a specific code fragment (i.e. documentation, comments, descriptions of algorithms) via a software repository similar to the one proposed for tool integration. Essentially, such a software repository can be part of the compiler. The front-end can be used to populate the repository and the debugger provide dynamic information on specific execution traces.

The second possible avenue of research is to use Markov models to represent the structure and the dependencies between high level software architectural patterns. In such a way multiple high level architectural decomposition views of a large software system can be revealed, based on a set of high level architectural descriptions encoded in an abstract architecture description language much like ACL. The software architect may provide a number of queries that extract architectural views between high level components based on their top level interaction, dependencies, and organizational patterns.

Our experience with the proposed tools is that they can easily provide information on differences between software versions, modifications between releases, and can be used for system partitioning. As a result, all three discussed techniques have been integrated into our Re-engineering Environment and are part of our standard software analysis tool-set.

CHAPTER 9

Bibliography

REFERENCES

[Adamov87] Adamov, R. "Literature review on software metrics", *Zurich: Institut fur Informatik der Universitat Zurich*, 1987.

[Aho89] Aho, A., Ganapathi, M., Tjiang, S., "Code Generation Using Tree Matching and Dynamic Programming" *ACM Transactions on Programming Languages and Systems*, vol. 11, No. 4, October 1989, Pages 491-516.

[Aho85] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1985.

[Albrecht79] Albrecht, A., J., "Measuring Application Development Productivity", *Proceedings of IBM Applications Development Symposium*, Monterey, CA., Oct. 1979, pp.83-92.

[Anderberg73] Anderberg M., "Cluster Analysis for Applications" *Academic Press.*

[Baker95] Baker S. B, "On Finding Duplication and Near-Duplication in Large Software Systems" *In Proceedings of the Working Conference on Reverse Engineering 1995*, Toronto ON. July 1995i, pp. 86-95.

[Baker94] Baker S. B, "Parameterized Pattern Matching: Algorithms and Applications", *Journal Computer and System Sciences*, 1994.

[Beneduci89] Beneducci, A., "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance," *IEEE Conf. on Software Maintenance*, 1989, pp. 180.

[Bental92] Bental D. "Using Clausal Join and Clausal Split to Recognize Language Specific Programming Design Decisions", *Workshop Notes, AI and Automated Program Understanding, Conference of the American Association of Artificial Intelligence* 1992, pp. 37 - 40.

[Biggerstaff89] Biggerstaff, T. J., "Design Recovery for Maintenance and Reuse," *IEEE Computer*, July 1989, pp. 36-48.

[Biggerstaff94] Biggerstaff, T., Mitbander, B., Webster, D., "Program Understanding and the Concept Assignment Problem", *Communications of the ACM*, May 1994, Vol. 37, No.5,

[Bmc] http://www.beirut.bmc.com pp. 73-83.

[Brown92] P. Brown et. al. "Class-Based n-gram Models of natural Language", *Journal of Computational Linguistics*, Vol. 18, No.4, December 1992, pp.467-479.

[Brotsky84] Brotsky, D.C., "An Algorithm for Parsing Flow Graphs," Master's thesis, MIT, 1984.

[Bush85] Bush, "The Automatic Restructuring of COBOL," *IEEE Conf. on Software Maintenance*, 1985, pp. 35-42.

[Buss94] E. Buss, R. De Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster and K. Wong, "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project", IBM Systems Journal, vol. 33 no. 3, 1994, pp. 477-500.

[Cadr] http://www.bicsystems.com/developer/cadre.html

[Catell49] Catell R., "R_D and other coefficients of pattern similarity", *Psychometrika 14*, pp. 279-288.

[Clifford75] Clifford H., Stephenson, W., "An Introduction to Numerical Classification" *Academic Press, Inc., New York.*

[Clips] C-Language Integrated Production System *User's Manual* NASA Software Technology Division, Johnson Space Center, Houston, TX.

[Callics88] Callics, "A Knowledge Based System for Software Maintenance," *IEEE Conf. on Software Maintenance*, 1988, pp. 319-326.

[Canfora94] Canfora, G., Cimitile, A., DeLucca, A., "Software Salvaging Based on Conditions" *IEEE Conf. on Software Maintenance*, 1994, pp. 424-433.

[Canfora92] Canfora, G., Cimitile, A., Carlini, U., "A Logic-Based Approach to Reverse Engineering Tools Production" *Transactions of Software Engineering*, Vol.18, No. 12, December 1992, pp. 1053-1063.

[CASE89] "Re-engineering and Maintenance," *CASE Outlook 89*, No 3, 1989.

[Chakrabarti86] Chakrabarti, P., Ghose, S. and DeSarkoz, S., "Heuristic Search Through Islands," *AI Magazine*, Vol. 29, 1986, pp. 339 - 347.

[Chien91] Chien, J-H., Fu, S-T., Horowitz, E. and Rouff, C., "RPP: A System for Prototyping User Interfaces," 1991, pp. 419 - 420.

[Chiko90] Chikofsky, E.J. and Cross, J.H. II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan. 1990, pp. 13 - 17.

[Choi90] Choi, S.C. and Scacchi, W., "Extracting and Restructuring the Design of Large Systems," *IEEE Software*, Jan 1990, pp. 66 - 71.

[Cimitile90] Cimitile, A., Lucca, G. and Maresca, P., "Maintenance and Intermodular Dependencies in Pascal Environments," *1990 IEEE Conference on Software Maintenance*, 1990, p. 72.

[Colbrook89] Colbrook, "The Retrospective Introduction of Abstraction into Software," *IEEE Conf. on Software Maintenance*, 1989, pp. 166-173.

[Corazza90] Corazza, A., De Mori, R., Gretter, R. and Satta G., "Computation of Probabilities for an Island-Driven Parser," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Sept. 1991.

[Dart87] Dart, S.A., Ellison, R.J., Feiler, P.H. and Habermann, A.N., "Software Development Environments," *IEEE Computer*, Nov. 1987, pp. 18 - 27.

[Das89] Das, "A Knowledge Based Approach to the Analysis of Code and Program Design Language," *1989 IEEE Conference on Software Maintenance*, pp. 290-296.

[Davies90] Davies, S., "The Nature and Development of Programming Plans," *International Journal of Man Machine Studies*, Vol. 32, 1990, pp. 461 - 481.

[DeMori89] De Mori, R. and Prager, R., "Perturbation Analysis with Qualitative Models," *Proceedings of IJCAI 1989.*

[Dettienne90] Dettienne, F. and Soloway, E., "An Empirically Derived Control Structure for the Process of Program Understanding," *International Journal of Man Machine Studies*, Vol. 33, 1990, pp. 323 - 342.

[Edmonds82] Edmonds, E.D., "The Man-Computer Interface: A note on Concepts and Design," *International Journal of Man-Machine studies*, no. 16, 1982, pp. 231 - 236.

[Engberts91] Engberts, A., Kozaczynski, W., Ning, J. "Automating software maintenance by concept recognition-based program transformation," *IEEE Conference on Software Maintenance - 1991*, IEEE, IEEE Press, October 14-17 1991.

[Everitt74] Everitt B., "Cluster Analysis" *John Wiley & Sons, Inc*, New York.

[Fickas79] Fickas, S., Brooks, R., "Recognition in a program understanding system". In Proc. *6th Int. Joint Conf. Artificial Intelligence*, Tokyo, Japan, 1979, pp. 266-268.

[Fenton91] Fenton, E. "Software metrics: a rigorous approach", Chapman and Hall, 1991.

[Gallagher91] Gallagher, K.B. and Lyle, J.R., "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, August 1991, pp. 751-761.

[Gillis90] Gillis, K. and Wright, D., "Improving Software Maintenance Using System Level Reverse Engineering," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 84-91.

[Green88] Green, "Self Identifying Software," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 126-133.

[Guedj80] Guedj, R.A. et al., *Methodology of Interaction: Seillac II (Seillac, France)*, Amsterdam, 1980.

[Hale90] Hale, D., Haworth, D. and Sharpe, S., "Empirical Software Maintenance Studies during the 1980's," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 118-125.

[Halst77] Halstead, M., H., "Elements of Software Science", New York: Elsevier North-Holland, 1977.

[Hanau80] Hanau, R. and Lenorovitch, R., "Prototyping and Simulation Tools for User/Computer Dialogue Design," *Proceedings of the ACM SIGRAPH 80 7th Annual Conference on Computer Graphics and Interactive Techniques*, Seattle Wash., 1980.

[Harandi88] Harandi, "PAT: A Knowledge Based Program Analysis," *IEEE Conf. on Software Maintenance*, 1988, pp. 312-319.

[Harandi90] Harandi, M.T. and Ning, J.Q., "Knowledge-Based Program Analysis," *IEEE Software*, Jan 1990, pp. 74 - 81.

[Hartman92] Hartman, J., "Technical Introduction to the First Workshop on AI and Automated Program Understanding", San Jose 1992. *AAAI'92 Workshop on Automated Program Understanding and Artificial Intelligence*

[Hartman91a] Hartman, J., "Automatic Control Understanding for Natural Programs," University of Texas at Austin, PhD., May 1991.

[Hartman91b] Hartman, J., "Understanding Natural Programs Using Proper Decomposition," *Proceedings of the 13th International Conference of Software Engineering*, May 1991.

[Hartigan75] Hartigan J., "Clustering Algorithms" John Wiley & Sons, 1975

[Hartson89] Hartson, H. and Hix, D., "Toward Empirically Derived Methodologies and Tools for Human Computer Interface Development," *International Journal of Man Machine Studies*, Vol. 31, 1989, pp. 477 - 494.

[Hausler90] Hausler, P., et.al "Using Function Abstraction to Understand Program Behavior" *IEEE Software, January 1990*, pp. 55-63.

[Henderson87] Henderson, P.B. and Notkin, D., "Integrated Design and Programming Environments," *IEEE Computer*, Nov 1987, pp. 12 - 16.

[Hennessy91] Hennessy M., "The Semantics of Programming Languages : An Elementary Introduction using Structural Operational Semantics", Wiley 1991.

[Henry81] Henry, S., Kafura, D., Harris, K., "On the Relationships among the Three Software Metrics", *Proceedings of 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*, March 1981.

[Hill87] Hill, "Event-response Systems: A Technique for Specifying Multithreaded Dialogues," *Proceedings of the ACM CHI + GI Conference*, 1987, pp. 241 - 248.

[Hix89] Hix, D. and Hartson, R., "Human-Computer Interface Development: Concepts and Systems for its Management," *ACM Computing Surveys*, Vol. 21, March 89, pp. 5 - 92.

[Hoare85] Hoare, C.A.R., "Communicating Sequential Processes," *Series in Computer Science*, Prentice-Hall International, London, 1985.

[Holland89] Holland, "Tools for Preventing Software Maintenance," *1989 IEEE Conference on Software Maintenance*, 1989, pp. 2-9.

[Hopcroft79] Hopcroft, J., Ullman, J., "Introduction to Automata Theory, Languages, and Computation" *Addison Wesley*, 1979.

[Huff89] Huff, K. and Lesser, V., "A Plan Based Intelligent Assistant that Supports the Software Development Process," *ACM SIGSOFT/SIGPLAN Third Symposium on Software Engineering Environments*, Boston, MA, Nov. 1988, pp. 97 - 106.

[HypSoft] http://www.hypersoft.co.uk

[Intersolv91] *Design Recovery for Excelerator*, Intersolv Sales Brochure, 1991.

[Intersolv] http://www.intersolv.com

[Jain88] Jain. A, Dubes. R., "Algorithms for Clustering Data" Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Jankowitz88] Jankowitz, H., T., "Detecting Plagiarism in student PASCAL programs". *Computer Journal*, 31(1):1-8, 1988.

[Johnson85] Johnson, W.L. and Soloway, E., "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering*, March 1985, pp. 267 - 275.

[Johnson94a] Johnson, H., "Substring Matching for Clone Detection and Change Tracking", *International Conference on Software Maintenance* 1994, Victoria BC, 21-23 September, 1994, pp.120-126.

[Johnson94b] Johnson, H., "Visualizing Textual Redundancy in Legacy Source", In Proceedings of the 1994 *IBM NRC CAS Conference (CASCON '94)*, Toronto, Ontario, October 31 - November 3, 1994, pp.9-18.

[Jones81] Jones, K., "Information Retrieval Experiment" Butterworths Publishing Co., Toronto, 1981.

[Kaplan73] Kaplan, R., "A General Syntactic Processor," in *Natural Language Processing*, ed: Rustin E.,, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Karakostas90] Karakostas, V., "The Use of Application Domain Knowledge for Effective Software Maintenance," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 170-178.

[Kay80] Kay, M., *Algorithm Schemata and Data Structures in Syntactic Processing*, Xerox, Palo Alto research Center, 1980.

[Kenning90] Kenning, R. and Munro, M., "Understanding the Configurations of Operational Systems," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 20-28.

[Ketabchi90] Ketabchi, M., "An Object Oriented Integrated Software Analysis and Maintenance," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 60.

[Konto96a] Kontogiannis K., DeMori, R., Merlo, E., Galler, M., Bernstein, M., "Pattern Matching for Clone and Concept Detection", Journal of Automated Software Engineering, vol.3, 1996, pp.77-108.

[Konto96b] Kontogiannis K., Mylopoulos J., Stanley, M., "Experiences on Migrating Procedural Systems to Object Oriented Architectures" *OOPSLA'96 Workshop on Transforming Legacy Systems to Object Oriented Systems*, San Jose Ca., 1996.

[Konto95] Kontogiannis K., DeMori, R., M., Bernstein, Merlo, E., Galler, M. "Pattern Matching for Design Concept Localization" In Proceedings of WCRE'95 pp. 96-103, July, 14-16, Toronto, Canada.

[Konto94] Kostas Kontogiannis, Renato DeMori, Morris Bernstein and Ettore Merlo, "Localization of Design Concepts in Legacy Systems". In the Proceedings of the ICSM'94, Victoria, Canada, pp.414-423.

[Kotik89] Kotik, G.B. and Markosian, L.Z., *Automating Software Analysis and Testing Using a Program Transformation System*, Reasoning Systems Inc., 1989.

[Kuhn90] Kuhn, R., DeMori, R., "A Cache-Based Natural Language Model for Speech Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.12, No.6, June 1990, pp.570-583.

[Landis88] Landis, "Documentation in a Software Maintenance Environment," *1989 IEEE Conference on Software Maintenance*, 1989, pp. 2-9.

[Lebowitz83] Lebowitz, M., "Memory-Based Parsing," *AI Magazine*, Vol. 21, 1983, pp. 363 - 403.

[Letovsky88] Letovsky, S. "Plan Analysis of Programs," *Ph.D thesis, Yale University, Dept. of Computer Science, YALEU/CSD/RR662*, December 1988.

[Lieberman84] Lieberman, H., "Seeing what your programs are doing," *International Journal of Man Machine Studies*, Vol. 21, pp. 311 - 331.

[Ligner88] Ligner, J., "Software Maintenance as Engineering Discipline," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 292-300.

[Lowry92] Lowry, M. R., Kedar, S. D., "Toward a Learning Apprentice for Software Re-engineering", *Workshop Notes, AI and Automated Program Understanding, AAAI'92* 1992, pp. 76 - 84.

[Lu78] Lu S., Fu, K., "A sentence-to-sentence clustering procedure for pattern analysis" *IEEE Transactions on Systems, man and Cybernetics*, SMC 8, pp. 381-389.

[Lutz89] Lutz, R., "Chart Parsing of Flowgraphs," *in Proceedings of International Joint Conference on Artificial Intelligence 89*, 1989, pp. 116 - 121.

[Mason81] Mason, R. and Carfy, T., "Productivity Experiences with a Scenerio Tool," *Proceedings of IEEE COMPCON*, 1981, pp. 106 - 111.

[Maarek91] Maarek Y., Berry, D., Kaiser, G., "An Information Retrieval Approach For Automatically Constructing Software Libraries", *IEEE Transactions in Software Engineering*, vol.17, No. 8, August 1991, pp.800-813.

[McCabe90] McCabe T., J., "Reverse Engineering, reusability, redundency: the connection" *American Programmer*, 3(10):8-13, Oct. 1990.

[McCabe76] McCabe T., J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol.7, No. 4, Sept. 1976, pp.308-320.

[MacLaugh95] McLaughlan, M., : M.Sc. Project Thesis "REVERSE Engineering PL/AS Software Using Metrics", McGill University, Department of Computer Science.

[Magdal96] Magdalinos, C., : M.Sc. Research Thesis "A Generic Stochastic Pattern Matcher for Plan Recognition in Software Systems", McGill University, Department of Computer Science.

[Mahala36] Mahalanobis, P. "On the generalized distance in statistics" *Proc. Indian Nat. Inst. Sci*, vol.2, pp.49-55 1936.

[Meekel88] Meekel, "LOGISCOPE: A Tool for Maintenance," *IEEE Conf. on Software Maintenance*, 1988, pp. 328-337.

[Merlo89] Merlo, E., " An Artificial Intelligence Language to Describe Extended Procedural Networks", *Ph. D. Thesis, McGill University*, Montreal, May 1989.

[Merlo93] Merlo, E., McAdam. I, De Mori. R., "Source Code Informal Information Analysis Using Connectionist Models", *International Joint Conference on Artificial Intelligence*, August 29 - September 3, 1993, Chambery, France, pp.1339-1344.

[Milner89] Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

[Muller91] Müller, H.A., "Rigi as a Reverse Engineering Tool", *Technical Report, Dept. of Computer Science*, University of Victoria, March 1991.

[Muller93] Müller, H.A., "Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project" *In Proceedings of CASCON'93*, Toronto, ON. 24-28 Oct. pp. 217-226.

[Mulligan91] Mulligan, "User Interface Design in the Trenches: Some Tips on Shooting from the Hip," *Human Factors in Computing Systems*, 1991, pp. 232-241.

[Murray88] Murray, W., "Automatic Program Debugging for Intelligent Tutoring Systems", *Morgan Kaufman*, San Matteo, CA, 1988.

[Myers86] Myers, B.A. and Buxton, W., "Creating Highly Interactive and Graphical User Interfaces by Demonstration," *SIGGRAPH '86*, pp. 249 - 258.

[Myers89] Myers, B.A., "User-Interface Tools: Introduction and Survey," *IEEE Software*, Jan 1989, pp. 15 - 23.

[Mylo96] Mylopoulos, J., Gal, A., Kontogiannis, K., Stanley, M., "A Generic Integration Architecture for Cooperative Information Systems" *in Proceedings of Co-operative Information Systems'96*, Brussels, Belgium, pp.208-217.

[Mylo90] Mylopoulos, J., "Telos : A Language for Representing Knowledge About Information Systems," *University of Toronto, Dept. of Computer Science Technical Report KRR-TR-89-1*, August 1990, Toronto.

[Ning94] Ning, J., Engberts, A., Kozaczynski, W., "Automated Support for Legacy Code Understanding", *Communications of the ACM*, May 1994, Vol.37, No.5, pp.50-57.

[Norman84] Norman, D.A., "Four stages of User Activities," *Proceedings of INTERACT '84, First IFIP Conference on Human-Computer Interaction*, 1984.

[Olsen87] Olsen, D.R.Jr., "Whither UIMS?," *Proceedings from the conf. on Human Factors in Computing Systems and Graphics Interface*, 1987, pp. 311 - 314.

[Osborne90] Osborne, D., "Fitting Pieces to the Maintenance Puzzle," *IEEE Software*, Jan. 1990, pp. 11-21.

[Overstreet88] Overstreet, "Program Maintenance by Safe Transformations," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 118-125.

[Ourston89] Ourston, D., "Program Recognition," *IEEE Expert*, Winter 1989, pp. 36.

[Ovum90] Rock-Evans, R. and Hales, K., *Reverse Engineering: Markets, Methods and Tools*, Ovum Ltd., 1990.

[Paul94] Paul, S., Prakash, A., "A Framework for Source Code Search Using Program Patterns", *IEEE Transactions on Software Engineering*, June 1994, Vol. 20, No.6, pp. 463-475.

[Pearson26] Pearson, K., "On the coefficient of racial likeness" *Biometrika*, vol.18, p. 105.

[Plotkin81] Plotkin G. D, "Structural Operational Semantics", *Lecture notes, DAIMI FN-19, Aarhus University, Denmark*, 1981.

[Prieto-Diaz90] Prieto-Diaz, R., "Domain Analysis: An Introduction," *Software Engineering Notes*, Vol. 15, No. 2, April 1990, pp. 47 - 54.

[Quilici96] Quilici "Reengineering of Legacy Systems: Is it Doomed to Failure?", *International Conference on Software Engineering*, Berlin 1996

[Quilici92] Quilici, A., Khan, J. "Extracting Objects and Operations from C Programs, " *Workshop Notes, AI and Automated Program Understanding, AAAI'92* 1992, pp. 93 - 97.

[Quilici94] Quilici, A., "A Memory Based Approach to Recognizing Programming Plans" *Communications of the ACM, May 1994, vol.37, No.5*, pp. 84-93.

[Rajlich88] Rajlich, V., "Visual Support for Programming in the Large," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 92-100.

[Reiss84] Reiss, S.P., "Pecan: Program Development systems that support Multiple Views," *ICSE-7*, 1984, pp. 324 - 333.

[Ryder89] Ryder, "ISMM: The Incremental Software Maintenance Manager," *1989 IEEE Conference on Software Maintenance*, 1989, pp. 142-150.

[Rich89] Rich, C. and Waters, R.C., *Intelligent Assistance for Program Recognition, Design, Optimization, and Debugging*, Memo, MIT AI Lab, Jan 1989.

[Rich90] Rich, C. and Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan 1990, pp. 82 - 89.

[Rugaber90] Rugaber, S., Ornburn, S.B. and LeBlanc, R.J. Jr., "Recognizing Design Decisions in Programs," *IEEE Software*, Jan 1990, pp. 46 - 54.

[Scott76] Scott, D., "Data Types as Lattices," *SIAM Journal of Computing*, Vol. 5, No 3, 1976, pp. 522 - 587.

[Selby90] Selby, R., Basili, V., "Analyzing Error Prone System Structure" *IEEE Transactions on Software Engineering*, vol 17, No. 2, February, 1991, pp. 141 - 152.

[Shepard79] Shepard, R., Carrol, J., "Additive Clustering:representation of similarities as combinations of discrete overlapping properties" *Psychological Review, 86* pp.87-123.

[Shneiderman86] Shneiderman, B., *Designing the User Interface*, Addison-Wesley Publishing co., 1986.

[Smythe90] Smythe, C., Colbrook, A. and, Darlison A., "Data Abstraction in a Software Reengineering Reference Model," *1990 IEEE Conference on Software Maintenance*, 1990, pp. 2-9.

[Sneath73] Sneath, P., Sokal, R., "Numerical Taxonomy", W.H. Freeman and Co., Publishers, san Fransisco

[Sneed87] Sneed, H.M., "Software Recycling," *1987 IEEE Conference on Software Maintenance*, 1987, pp. 82.

[Sneed88] Sneed, H.M. and Jandrasics, G. "Inverse Transformation of Software from Code to Specification," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 102-109.

[Spath80] Spath H., "Cluster Analysis Algorithms for Data Reduction and Classification of Objects", Ellis Horwood Publishers, West Sussex, England

[Stallings91] Stallings, W., "Data and Computer Communications" *MacMillan Inc.* Toronto, 1991

[Stoy77] Stoy, J.E., *Denotational Semantics*, MIT Press, 1977.

[Tilley95] Tilley, S.R.; K. Wong; M.-A.D. Storey; and H.A. Müller. "Programmable Reverse Engineering," *International Journal of Software Engineering and Knowledge Enginering*, Vol. 4, No. 4, pp. 501-520, December 1994.

[Vite67] Viterbi, A.J, "Error Bounds for Convolutional Codes and an Asymptotic Optimum Decoding Algorithm", *IEEE Trans. on Information Theory*, 13(2) 1967.

[Ward89] Ward, M., Calliss, F.W. and Munro, M., "The Maintainer's Assistant," *IEEE Conf. on Software Maintenance*, 1989, pp. 307 - 313.

[Wasser90] Wasserman, A., "Tool Integration in Software Engineering Environments" *Lecture Notes in Computer Science* 467, Sprienger-Verlag, Berlin, pp. 138-150

[Wedo85] Wedo, "Structured Program Analysis Applied to Software Maintenance," *IEEE Conf. on Software Maintenance*, 1985, pp. 28-36.

[Whitfield91] Whitfield D., Soffa M. L., "Automatic Generation of Global Optimizers", *1991 ACM SIGPLAN, Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 26-28, 1991.

[Wild88] Wild, C. and Maly, K., "Towards a Software Maintenance Support Environment," *1988 IEEE Conference on Software Maintenance*, 1988, pp. 80 - 85.

[Wild89] Wild, C., Maly, K., Liu, L., Chen, J. and Xu, T., "Decision-Based Software Development: Design and Maintenance," *IEEE Conf. on Software Maintenance*, 1989, pp. 297 - 306.

[Wilde89] Wilde, "Dependency Analysis Tools: Reusable Components for Software Maintenance," *1989 IEEE Conference on Software Maintenance*, 1989, pp. 126-133.

[Wills93] Wills, L.M., "Automated Program Recognition by Graph Parsing" *MIT Technical Report 1358*, MIT, AI Laboratory, 1993

[Wills92] Wills, L.M., "Automated Program Recognition: Breaking out of the Toy Program Rut, " *Workshop Notes, AI and Automated Program Understanding, AAAI'92* 1992, pp. 129 - 133.

[Wills87] Wills, L.M., "Automated Program Recognition," Master's thesis, MIT, 1987.

[Wills90] Wills, L.M., "Automated Program Recognition: A Feasibility Demonstration," *Artificial Intelligence*, Vol. 45, No. 1-2, Sept. 1990, pp.113-172.

[Viasoft] http://www.viasoft.com

[Xcessory] Report on *Xcessory*, Integrated Computer Solutions, Cambridge MA.

[Zupan82] Zupan J., "Clustering of Large Data Sets" Research Studies Press, England

APPENDIX A

This Appendix describes the Abstract Syntax Grammar for ACL. Left hand sides of the rules represent object classes for which instances are created each time a rule succeeds. This is the way the AST nodes are created. In the left hand side of each rule, attributes that are annotations to the object class at the head of the rule are used to connect AST nodes. This is the way the ACL AST is formed. Attributes and object classes are described in more detail in Appendix B.

```
        #|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
        #                        ACL  GRAMMAR                               #
        #|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
grammar PATTERN-ACL
 PRODUCTIONS
              ########################################
              #          TOP LEVEL OBJECTS           #
              ########################################
    Pattern-Object   ::= ["#" {["id-plan" ":" the-plan-name]} statement-in-pattern * ";" "#" ]
    Include-Object   ::= ["include" included-pattern-file included-plan-name]
    User-Plan-Object ::= ["plan:" user-plan-name]

              ########################################
              #         FUNCTION-DEF OBJECT          #
              ########################################
    Function-Def-Pattern ::= ["function" function-name-in-pattern  function-def-body-pattern ]

              ########################################
              #         GENERIC  STATEMENTS          #
              ########################################
   Generic-Statement-Pattern        ::= [ {overall-abstr-descr} statement-item]
   Composite-XOR-Statement-Pattern ::= ["(" composite-XOR-statement-item + "+" ")"]
```

```
Composite-Interleaved-Statement-Pattern ::= ["(" composite-Interleaved-statement-item + "-" ")"]
Any-Statement-Pattern ::= ["any-statement" [([any-statement-body-pattern !! is-complete?] |
                          any-statement-description)] ]
Pattern-*-Statement   ::= ["*-statement"  *-statement-pattern-description-item]
Pattern-1-Statement   ::= ["+-statement"  one-statement-pattern-description-item]

          ######################################
          #        ITERATIVE STATEMENTS        #
          ######################################
Iterative-Statement-Pattern ::= ["iterative-statement" "(" iterative-condition-pattern ")"
                                 iterative-body-pattern ]
While-Statement-Pattern     ::= ["while-statement"
                                "(" while-condition-pattern ")" while-body-pattern ]
Do-Statement-Pattern        ::= ["do-statement" "(" do-condition-pattern ")"
                                 do-body-pattern ]
For-Statement-Pattern       ::= ["for-statement" "(" for-initialize-pattern ";"
                                 for-test-pattern ";"
                                 for-increment-pattern ")"
                                 for-body-pattern ]

          ######################################
          #      CONDITIONAL STATEMENTS        #
          ######################################
Conditional-Statement-Pattern ::= ["conditional-statement" "(" conditional-cond-pattern ")"
                                   conditional-body-pattern]
If-Statement-Pattern     ::= ["if-statement" "(" if-condition-pattern ")" "then"
                            then-pattern {["else"  else-pattern  ]}]
Switch-Statement-Pattern ::= ["switch-statement"  "(" switch-pattern ")"
                            switch-body-pattern  ]

          ######################################
          #         BASIC  STATEMENTS          #
          ######################################

Block-Statement-Pattern      ::= ["{" block-statements-in-pattern * ";" "}"]
Labelled-Statement-Pattern   ::= ["labelled-statement" labelled-pattern]
Return-Statement-Pattern     ::= ["return-statement" "(" return-pattern ")"]
```

```
GoTo-Statement-Pattern       ::= ["goto-statement" "(" goto-pattern ")"]
Continue-Statement-Pattern   ::= ["continue"]
Break-Statement-Pattern      ::= ["break"]
Expression-Statement-Pattern ::= [expression-statement-pattern-body]

          ########################################
          #         OVERALL DESCRIPTORS          #
          ########################################
Overall-Pattern-Description ::= ["overall-description"
                                 overall-pattern-description-feature-item + ","]

          ########################################
          #        STATEMENT DESCRIPTORS         #
          ########################################
Detailed-Pattern-Description ::= ["abstract-expression-description"
                                 ([expression-pattern-description-feature-item + ","] |
                                  expression-empty-item) ]

          ########################################
          #     FEATURE VECTORS DESCRIPTORS      #
          ########################################
Feature-Item ::= [description-vector + ","]

          ########################################
          #            FEATURES                  #
          ########################################
Empty-Description       ::= ["empty"]
Keywords-Description    ::= ["keywords :"  "[" keywords-in-pattern + "," "]" ]
Defines-Description     ::= ["defines :"  "[" definitions-in-pattern + "," "]" ]
Uses-Description        ::= ["uses :"  "[" uses-in-pattern  + "," "]"]
Probability-Description ::= ["probability :" "[" probabilty-tuple + "," "]"]
Probability-Tuple-Item  ::= [stat-name stat-probability-value]
Metrics-Description     ::= ["metrics :" "[" metrics-in-pattern + "," "]"]

          ########################################
          #       ABSTRACT IDENTIFIERS           #
          ########################################
```

```
Bind-Variable-Object      ::= [(["?" var-name] | [actual-var-name !! is-actual-var?])
                               {bind-variable-type} ]
Bind-Variable-Type-Object ::= [":" {(["*" !! is-pointer? ] |
                                    ["&" !! is-reference?])} bind-type]

          ######################################
          #       ABSTRACT DATA TYPES          #
          ######################################
Abstract-Type-Object ::= ["^" (["numeral" !! is-numeral?] |
                               ["char" !! is-char?] |
                               [ "struct" !! is-struct-type?] |
                               ["void-type" !! is-void?] |
                               ["enum-type" !! is-enum?] |
                               ["array-type" !! is-array?] |
                               ["any-type" !! is-any-type?] |
                               [the-type-name !! is-actual-name?]) ]

          ######################################
          #       ABSTRACT EXPRESSIONS         #
          ######################################
Assignment-Pattern          ::= ["assignment-statement" assignment-pattern-description-item]
Actual-Assignment-Patterni  ::= ["actual-assignment-statement"
                                  actual-assignment-pattern-description-item]
PostIncrementation-Pattern  ::= ["postincrementation-statement" postinc-pattern-description-item]
PostDecrementation-Pattern  ::= ["postdecrementation-statement" postdec-pattern-description-item]
PreDecrementation-Pattern   ::= ["predecrementation-statement" predec-pattern-description-item]
PreIncrementation-Pattern   ::= ["preincrementation-statement" preinc-pattern-description-item]
Function-Call-Pattern       ::= ["function-call" function-call-name
                                  function-call-pattern-description-item]
Condition-Pattern           ::= [(["equality" !! is-equality-cond?] |
                                  ["inequality" !! is-inequality-cond?] |
                                  ["fcn-call-test" !! is-fcn-call-test-cond? ] |
                                  ["boolean-test" !! is-boolean-test-cond?] |
                                  ["any-cond" !! is-any-cond?]) the-condition-description]
```

APPENDIX B

In this appendix the Domain model of the ACL language is provided for further reference. Object Classes are denoted with capital letters, and correspond to the object classes at the heads ofthe Grammar Rules described in Appendix A. Also note that *map* constructs denote an AST attribute that is used to link AST object nodes represented by instances of the particular object classes. An object instance is created every time a ACL grammar rule succeds.

```
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
#                  ACL   DOMAIN   MODEL                                     #
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ########################################
              #          TOP LEVEL OBJECTS           #
              ########################################
var PATTERN-OBJECT           : object-class subtype-of Reverse-Engineering-Object
var the-plan-name            : map(Pattern-Object, symbol)
var pattern-description-item : map(Pattern-Object, Detailed-Pattern-Description)
var function-name-in-pattern : map(Pattern-Object, string)
var statement-in-pattern     : map(Pattern-Object, seq(Generic-Statement-Pattern))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var INCLUDE-OBJECT : object-class subtype-of Statement-Pattern
var included-pattern-file : map(Include-Object, symbol)
var included-plan-name    : map(Include-Object, symbol)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var USER-PLAN-OBJECT      : object-class subtype-of Statement-Pattern
var user-plan-name        : map(User-Plan-Object, symbol)
var user-plan-ast-root    : map(User-Plan-Object, Pattern-Object)
var plan-included-object  : map(User-Plan-Object, Include-Object)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ########################################
```

```
                #        FUNCTION-DEF OBJECT         #
                ######################################
var FUNCTION-DEF-PATTERN      : object-class subtype-of Statement-Pattern
var function-def-body-pattern : map(Function-Def-Pattern, Block-Statement-Pattern)


#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
                ######################################
                #        GENERIC  STATEMENTS         #
                ######################################
var GENERIC-STATEMENT-PATTERN : object-class subtype-of Statement-Pattern
var overall-abstr-descr       : map(Generic-Statement-Pattern,
                                    Overall-Pattern-Description)
var statement-item            : map(Generic-Statement-Pattern, Statement-Pattern)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var COMPOSITE-XOR-STATEMENT-PATTERN : object-class subtype-of  Basic-Statement-Pattern
var composite-XOR-statement-item    : map(Composite-XOR-Statement-Pattern,
                                          seq(Statement-Pattern))

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var COMPOSITE-INTERLEAVED-STATEMENT-PATTERN : object-class subtype-of  Basic-Statement-Pattern
var composite-Interleaved-statement-item    : map(Composite-XOR-Statement-Pattern,
                                                  seq(Statement-Pattern))

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var ANY-STATEMENT-PATTERN      : object-class subtype-of Basic-Statement-Pattern
var any-statement-body-pattern : map(Any-Statement-Pattern, Block-Statement-Pattern)
var any-statement-description  : map(Any-Statement-Pattern,
                                     Detailed-Pattern-Description)
var is-complete?               : map(Any-Statement-Pattern, Boolean)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PATTERN-*-STATEMENT                    :  object-class subtype-of Statement-Pattern
var *-statement-pattern-description-item   : map(Pattern-*-Statement,
                                                 Detailed-Pattern-Description)
var used?                                  : map(Pattern-*-Statement, Boolean)
```

```
#||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PATTERN-1-STATEMENT                    :  object-class subtype-of Statement-Pattern
var one-statement-pattern-description-item : map(Pattern-1-Statement,
                                                 Detailed-Pattern-Description)
#||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ######################################
              #        ITERATIVE STATEMENTS        #
              ######################################
var ITERATIVE-STATEMENT-PATTERN        : object-class subtype-of  Basic-Statement-Pattern
var iterative-condition-pattern        : map(Iterative-Statement-Pattern, Condition-Pattern)
var iterative-body-pattern             : map(Iterative-Statement-Pattern, Statement-Pattern)
var iterative-pattern-description-item : map(Iterative-Statement-Pattern,
                                             Detailed-Pattern-Description)

#||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var WHILE-STATEMENT-PATTERN        : object-class subtype-of  Iterative-Statement-Pattern
var while-condition-pattern        : map(While-Statement-Pattern, Condition-Pattern)
var while-body-pattern             : map(While-Statement-Pattern, Statement-Pattern)
var while-pattern-description-item : map(While-Statement-Pattern,
                                         Detailed-Pattern-Description)

#||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var DO-STATEMENT-PATTERN         : object-class subtype-of  Iterative-Statement-Pattern
var do-condition-pattern         : map(Do-Statement-Pattern,
                                       Condition-Pattern)
var do-body-pattern              : map(Do-Statement-Pattern,
                                       Statement-Pattern)
var do-pattern-description-item  : map(Do-Statement-Pattern,
                                       Detailed-Pattern-Description)

#||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var FOR-STATEMENT-PATTERN        : object-class subtype-of  Iterative-Statement-Pattern
var for-initialize-pattern       : map(For-Statement-Pattern, Statement-Pattern)
var for-test-pattern             : map(For-Statement-Pattern,
                                       Condition-Pattern)
var for-increment-pattern        : map(For-Statement-Pattern,
                                       Statement-Pattern)
```

```
var for-body-pattern             : map(For-Statement-Pattern, Statement-Pattern)
var for-pattern-description-item : map(For-Statement-Pattern,
                                       Detailed-Pattern-Description)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ######################################
              #      CONDITIONAL STATEMENTS        #
              ######################################
var CONDITIONAL-STATEMENT-PATTERN : object-class subtype-of Basic-Statement-Pattern
var conditional-cond-pattern      : map(Conditional-Statement-Pattern, Condition-Pattern)
var conditional-body-pattern      : map(Conditional-Statement-Pattern, Statement-Pattern)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var IF-STATEMENT-PATTERN          : object-class subtype-of  Conditional-Statement-Pattern
var if-condition-pattern          : map(If-Statement-Pattern, Condition-Pattern)
var then-pattern                  : map(If-Statement-Pattern, Statement-Pattern)
var else-pattern                  : map(If-Statement-Pattern, Statement-Pattern)
var then-pattern-description-item : map(If-Statement-Pattern, Detailed-Pattern-Description)
var else-pattern-description-item : map(If-Statement-Pattern, Detailed-Pattern-Description)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var SWITCH-STATEMENT-PATTERN : object-class subtype-of Conditional-Statement-Pattern
var switch-pattern           : map(Switch-Statement-Pattern, Condition-Pattern)
var switch-body-pattern      : map(Switch-Statement-Pattern, Statement-Pattern)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ######################################
              #         BASIC  STATEMENTS          #
              ######################################
var BLOCK-STATEMENT-PATTERN     : object-class subtype-of  Basic-Statement-Pattern
var block-statements-in-pattern : map(Block-Statement-Pattern, seq(Generic-Statement-Pattern))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var LABELLED-STATEMENT-PATTERN  : object-class subtype-of Statement-Pattern
var labelled-pattern : map(Labelled-Statement-Pattern, Detailed-Pattern-Description)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var RETURN-STATEMENT-PATTERN    : object-class subtype-of Basic-Statement-Pattern
var return-pattern : map(Return-Statement-Pattern, Detailed-Pattern-Description)
```

```
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var GOTO-STATEMENT-PATTERN      : object-class subtype-of Basic-Statement-Pattern
var goto-pattern : map(GoTo-Statement-Pattern,
                                    Detailed-Pattern-Description)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var CONTINUE-STATEMENT-PATTERN  : object-class subtype-of Basic-Statement-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var BREAK-STATEMENT-PATTERN     : object-class subtype-of Basic-Statement-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var EXPRESSION-STATEMENT-PATTERN : object-class subtype-of Basic-Statement-Pattern
var expression-statement-pattern-body : map(Expression-Statement-Pattern, Expression-Pattern)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
             ######################################
             #         OVERALL DESCRIPTORS        #
             ######################################
var OVERALL-PATTERN-DESCRIPTION              : object-class subtype-of Pattern-Description
var overall-pattern-description-feature-item : map(Overall-Pattern-Description, seq(Feature-Item))
var overall-empty-item                       : map(Overall-Pattern-Description,
                                                    Empty-Description)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
             ######################################
             #        STATEMENT DESCRIPTORS       #
             ######################################
var DETAILED-PATTERN-DESCRIPTION               : object-class subtype-of Expression-Pattern
var expression-pattern-description-feature-item : map(Detailed-Pattern-Description, seq(Feature-Item))
var expression-empty-item                      : map(Detailed-Pattern-Description,
                                                   Empty-Description)
var expanded-from                              : map(Detailed-Pattern-Description,
                                                       Statement-Pattern)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
             ######################################
             #     FEATURE VECTORS DESCRIPTORS    #
             ######################################
var FEATURE-ITEM       : object-class subtype-of Pattern-Object
```

```
var description-vector : map(Feature-Item, seq(A-Description-Object))

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
          ######################################
          #              FEATURES              #
          ######################################
var EMPTY-DESCRIPTION     : object-class subtype-of A-Description-Object
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var KEYWORDS-DESCRIPTION : object-class subtype-of A-Description-Object
var keywords-in-pattern   : map(Keywords-Description, set(string))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var DEFINES-DESCRIPTION   : object-class subtype-of A-Description-Object
var definitions-in-pattern : map(Defines-Description, set(Bind-Variable-Object))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var USES-DESCRIPTION      : object-class subtype-of A-Description-Object
var uses-in-pattern       : map(Uses-Description, set(Bind-Variable-Object))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PROBABILITY-DESCRIPTION : object-class subtype-of A-Description-Object
var probabilty-tuple        : map(Probability-Description, seq(Probability-Tuple-Item))
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PROBABILITY-TUPLE-ITEM  : object-class subtype-of Pattern-Object
var stat-name               : map(Probability-Tuple-Item, string)
var stat-probability-value  : map(Probability-Tuple-Item, real)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var METRICS-DESCRIPTION     : object-class subtype-of A-Description-Object
var metrics-in-pattern      : map(Metrics-Description, seq(real))

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
          ######################################
          #       ABSTRACT IDENTIFIERS         #
          ######################################
var BIND-VARIABLE-OBJECT  : object-class subtype-of Reverse-Engineering-Object
var var-name              : map(Bind-Variable-Object, symbol)
var is-actual-var?        : map(Bind-Variable-Object, Boolean)
var actual-var-name       : map(Bind-Variable-Object, symbol)
var bind-variable-type    : map(Bind-Variable-Object, Bind-Variable-Type-Object)
var var-bind              : map(Bind-Variable-Object, seq(C-Object))
```

```
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var BIND-VARIABLE-TYPE-OBJECT : object-class subtype-of Reverse-Engineering-Object
var bind-type                 : map(Bind-Variable-Type-Object, Abstract-Type-Object)
var is-pointer?               : map(Bind-Variable-Type-Object, Boolean)
var is-reference?             : map(Bind-Variable-Type-Object, Boolean)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ######################################
              #       ABSTRACT DATA TYPES          #
              ######################################
var ABSTRACT-TYPE-OBJECT : object-class subtype-of Reverse-Engineering-Object
var is-numeral?          : map(Abstract-Type-Object, Boolean)
var is-void?             : map(Abstract-Type-Object, Boolean)
var is-enum?             : map(Abstract-Type-Object, Boolean)
var is-array?            : map(Abstract-Type-Object, Boolean)
var is-char?             : map(Abstract-Type-Object, Boolean)
var is-struct-type?      : map(Abstract-Type-Object, Boolean)
var is-any-type?         : map(Abstract-Type-Object, Boolean)
var is-actual-name?      : map(Abstract-Type-Object, Boolean)
var the-type-name        : map(Abstract-Type-Object, symbol)

#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
              ######################################
              #       ABSTRACT EXPRESSIONS         #
              ######################################
var ASSIGNMENT-PATTERN          : object-class subtype-of Expression-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var ACTUAL-ASSIGNMENT-PATTERN   : object-class subtype-of Assignment-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var POSTINCREMENTATION-PATTERN  : object-class subtype-of Assignment-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PREINCREMENTATION-PATTERN   : object-class subtype-of Assignment-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var POSTDECREMENTATION-PATTERN  : object-class subtype-of Assignment-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var PREDECREMENTATION-PATTERN   : object-class subtype-of Assignment-Pattern
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
```

```
var FUNCTION-CALL-PATTERN       : object-class subtype-of Expression-Pattern
var function-call-name          : map(Function-Call-Pattern,
                                      Bind-Variable-Object)
var function-call-pattern-description-item
                                : map(Function-Call-Pattern,
                                      Detailed-Pattern-Description)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
var CONDITION-PATTERN           : object-class subtype-of Expression-Pattern
var the-condition-description   : map(Condition-Pattern, Detailed-Pattern-Description)
var is-equality-cond?           : map(Condition-Pattern, Boolean)
var is-inequality-cond?         : map(Condition-Pattern, Boolean)
var is-fcn-call-test-cond?      : map(Condition-Pattern, Boolean)
var is-boolean-test-cond?       : map(Condition-Pattern, Boolean)
var is-any-cond?                : map(Condition-Pattern, Boolean)
#|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||#
```

APPENDIX C

- **Example 1**: The following Plan is denoting instances of code fragments that implement the traversal of a linked list and the invocation of a *printf* function call when a condition related to the current node being traversed is met. Below the given Plan, we include the matched source code instances obtained from the Clips system.

```
actual-assignment-statement
        abstract-expression-description
        uses : [?currPtr : * ^ struct ],
        defines : [?ptr : * ^ struct];
*-statement
       abstract-expression-description empty;
 while-statement(any-cond
                 abstract-expression-description
                 uses : [?ptr : * ^ struct])
      {
         *-statement
                abstract-expression-description empty;
          if-statement(any-cond
                       abstract-expression-description
                       uses : [?ptr : * ^ struct] ) then
              {
                *-statement
                       abstract-expression-description empty ;
                function-call printf
                       abstract-expression-description
                       uses : [?ptr : * ^ struct];
                *-statement
                       abstract-expression-description empty
```

```
                        }
                      else
                        {
                          *-statement
                                abstract-expression-description empty
                        };
                    *-statement
                          abstract-expression-description empty
                }
- - - - - - - - - - - - - - - - - - - - -

#################################
# MATCHED SOURCE CODE INSTANCES #
#################################

field = o->ATTLIST[num].attFields;
while (field != ((void *)0))
          {
          if (!__strcmp(obj,origObj) ||
             (!__strcmp(field->AvalueType,"member") &&
              notInOrig ) )
              printf("[%s]\n",
                          field->Avalue);
          field = field->nextValue;
          }
FILE "object"
FROM-LINE : 2095
TO-LINE   : 2104
DIST      : 0.029003784
- - - - - - - - - - - - - - - - - - - -
field = o->ATTLIST[num].attFields;
while (field != ((void *)0))
          {
          if (!__strcmp(obj,origObj) ||
             (!__strcmp(field->AvalueType,"member") &&
              notInOrig ) )
              printf("[%s]\n",
```

```
                                field->Avalue);
            field = field->nextValue;
            }
FILE "object"
FROM-LINE : 2236
TO-LINE   : 2245
DIST      : 0.029003784
- - - - - - - - - - - - - - - - - - - - -
p = SEARCH_MEMBER(obj)->startList;
while(p != ((void *)0))
      {
       if (!__strcmp(p->type,"child"))
          {
           count = 1;
           printf("[%s]\n",p->link->name);
          }
       p = p->next;
      }
FILE "object"
FROM-LINE : 1283
TO-LINE   : 1292
DIST      : 0.010258662

- - - - - - - - - - - - - - - - - - - - -
p = SEARCH_SUB(obj)->startList;
while(p != ((void *)0))
      {
       if (!__strcmp(p->type,"child"))
          {
           count = 1;
           printf("[%s]\n",p->link->name);
          }
       p = p->next;
      }
FILE "object"
FROM-LINE : 1131
TO-LINE   : 1140
```

```
DIST      : 0.010258662
*****************************************
PLAN      : ll-search-and-print-1.pl
TIME-START : 10/04/96 08:00:58
TIME-END   : 10/04/96 09:03:11
TOTAL HITS : 3724
LEVEL OF ABSTR : 11557.0
MAX-DIST : 0.029003784
MIN-DIST : 0.010258662
AVG-DIST : 0.019631222
WEIGHT   : 0.5
RES-SIZE : 4
TIME-LOC-START : 10/04/96 08:00:59
TIME-LOC-END : 10/04/96 08:02:31
COVERAGE
"object" 1131 1140 0.010258662
"object" 1283 1292 0.010258662
"object" 2236 2245 0.029003784
"object" 2095 2104 0.029003784
*****************************************
```

- **Example 2**: The following Plan is denoting instances of code fragments that implement the addition of a new element to a linked list. Below the given Plan, we include the matched source code instances obtained from the Tcsh system.

```
actual-assignment-statement
            abstract-expression-description
            uses : [?head : * ^ struct ],
            defines : [?elem : * ^ struct],
            keywords : ["next"] ;
*-statement
          abstract-expression-description
          empty;
actual-assignment-statement
            abstract-expression-description
            uses : [?elem : * ^ struct ],
            defines : [?head : * ^ struct]
```

```
- - - - - - - - - - - - - - - - - - - - -

##################################
# MATCHED SOURCE CODE INSTANCES #
##################################

new->next = where->next;
where->next = new;
new->next->prev = new;

FILE "tc"
FROM-LINE : 1165
TO-LINE   : 1167
DIST      : 0.59725314
- - - - - - - - - - - - - - - - - - - - -

now->next->prev = now->prev;
free((ptr_t) now->word);
del = now;
now = now->next;
```

```
FILE "tc"
FROM-LINE : 1150
TO-LINE   : 1153
DIST      : 0.5256015
- - - - - - - - - - - - - - - - - - - - - -

now->prev->next = now->next;
now->next->prev = now->prev;
free((ptr_t) now->word);
del = now;
now = now->next;

FILE "tc"
FROM-LINE : 1149
TO-LINE   : 1153
DIST      : 0.43800125
- - - - - - - - - - - - - - - - - - - - - -

tmp->prev->next = tmp->next;
tmp->next->prev = tmp->prev;

FILE "tc"
FROM-LINE : 967
TO-LINE   : 968
DIST      : 0.37880763
- - - - - - - - - - - - - - - - - - - - - -

tmp->next->prev = tmp->prev;
free((ptr_t) tmp->word);
del = tmp;
tmp = tmp->next;

FILE "tc"
FROM-LINE : 960
TO-LINE   : 963
DIST      : 0.5256015
```

```
- - - - - - - - - - - - - - - - - - - - - -

tmp->prev->next = tmp->next;
tmp->next->prev = tmp->prev;
free((ptr_t) tmp->word);
del = tmp;
tmp = tmp->next;

FILE "tc"
FROM-LINE : 959
TO-LINE   : 963
DIST      : 0.43800125
- - - - - - - - - - - - - - - - - - - - - -

new1->next = w.prev = new2;
new1->prev = new2->next = &w;

FILE "tc"
FROM-LINE : 809
TO-LINE   : 810
DIST      : 0.98704046
- - - - - - - - - - - - - - - - - - - - - -

w.next = new2->prev = new1;
new1->next = w.prev = new2;
new1->prev = new2->next = &w;

FILE "tc"
FROM-LINE : 808
TO-LINE   : 810
DIST      : 0.7324082
- - - - - - - - - - - - - - - - - - - - - -

p = p->next;
if (any(RELPAR, p->word[0])) {
seterror(101);
continue;
```

```
    }
 if (((flags & 4) && (flags & 8) == 0) || t->R.T_drit)
seterror(102);
    else
t->R.T_drit = s_strsave(p->word);
 continue;
 case '<':
    if (l != 0)
goto savep;
 if (p->word[1] == '<')
t->t_dflg |= (1<<9);
 if (p->next == p2) {
seterror(101);
continue;
    }
 p = p->next;

 FILE "sh"
 FROM-LINE : 614
 TO-LINE   : 634
 DIST      : 0.099396266
 - - - - - - - - - - - - - - - - - - - - -

 retp->next = p2;
 p2->prev = retp;

 FILE "sh"
 FROM-LINE : 203
 TO-LINE   : 204
 DIST      : 0.29182288
 - - - - - - - - - - - - - - - - - - - - -

 alout.next->prev = p1;
 p1->next = alout.next;

 FILE "sh"
 FROM-LINE : 184
```

```
TO-LINE   : 185
DIST      : 0.34107885
- - - - - - - - - - - - - - - - - - - - -

alout.prev->prev->next = p1->next;
alout.next->prev = p1;
p1->next = alout.next;

FILE "sh"
FROM-LINE : 183
TO-LINE   : 185
DIST      : 0.53505206
- - - - - - - - - - - - - - - - - - - - -

p1->next->prev = alout.prev->prev;
alout.prev->prev->next = p1->next;
alout.next->prev = p1;

FILE "sh"
FROM-LINE : 182
TO-LINE   : 184
DIST      : 0.28240517
- - - - - - - - - - - - - - - - - - - - -

fp = vp->next;
vp->next = fp->next;

FILE "sh"
FROM-LINE : 235
TO-LINE   : 236
DIST      : 0.3080359
- - - - - - - - - - - - - - - - - - - - -

lp->next->prev = &np->Hlex;
np->Hlex.prev = lp->prev;

FILE "sh"
```

```
FROM-LINE : 106
TO-LINE   : 107
DIST      : 0.49954465
- - - - - - - - - - - - - - - - - - - - -

np->Hlex.next = lp->next;
lp->next->prev = &np->Hlex;
np->Hlex.prev = lp->prev;
lp->prev->next = &np->Hlex;

FILE "sh"
FROM-LINE : 105
TO-LINE   : 108
DIST      : 0.43944493
*****************************************

PLAN       : ll-add-elem-1.pl
TIME-START : 09/30/96 13:10:09
TIME-END   : 09/30/96 13:13:53
TOTAL HITS : 90
LEVEL OF ABSTR : 5043.0
MAX-DIST : 0.98704046
MIN-DIST : 0.099396266
AVG-DIST : 0.4637185
WEIGHT   : 0.5
RES-SIZE : 16
TIME-LOC-START : 09/30/96 13:10:16
TIME-LOC-END : 09/30/96 13:12:14
PLAN: ll-add-elem-1.pl
COVERAGE
"sh" 105 108 0.43944493
"sh" 106 107 0.49954465
"sh" 235 236 0.3080359
"sh" 182 184 0.28240517
"sh" 183 185 0.53505206
"sh" 184 185 0.34107885
"sh" 203 204 0.29182288
```

```
"sh" 614 634 0.099396266
"tc" 808 810 0.7324082
"tc" 809 810 0.98704046
"tc" 959 963 0.43800125
"tc" 960 963 0.5256015
"tc" 967 968 0.37880763
"tc" 1149 1153 0.43800125
"tc" 1150 1153 0.5256015
"tc" 1165 1167 0.59725314
```

APPENDIX D

In this Appendix we present sample queries used for the experiments presented in Chapter.6 We distinguish between two types of queries. The first type (*Code-to-Code* queries) represents queries used to obtain results for the Metrics-based and the DP-based approach. The second type represents ACL queries that were used to obtain results using the Markov-based approach.

- **Code-To-Code queries:**

```
for(i=0 ; i < or->numOfAtts ; i++)
   if (strcmp(o->ATTLIST[num].Aname,or->ATTLIST[i].Aname)==0)
      {
       notInOrig = 0;
       i = or->numOfAtts + 100;
      }
```

```
-------------------------------------------------------
for(num=0 ; num < p->numOfAtts ; num++)
      {
      field = p->ATTLIST[num].attFields;
      while (field != NULL)
      {
       printf("NAME  : %s\n VALUE : %s\n  CARDINALITY : %s\n   TYPE  : %s\n",
                          p->ATTLIST[num].Aname,
                          field->Avalue,
                          p->ATTLIST[num].Multiple,
                          field->AvalueType);
       printf("CAST : %s\n", field->AvalueCast);
       printf("\n");
       field = field->nextValue;
```

```
        }

  ------------------------------------
if(!strcmp(alist[num].Multiple,"s"))
      {
        putValueFunctionSingle(objName,alist[num].Aname,
        field->Avalue, field->AvalueType);
      }
        else
      {
        putValueFunctionMulti(objName,alist[num].Aname,
        field->Avalue, field->AvalueType);
      }

 ------------------------------------------
{
    if (ch != NULL)
     {
       lsp1 = startSub->startList;
       startSub->startList = getListNode();
       startSub->startList->next = lsp1;
       startSub->startList->link = ch;
       strcpy(startSub->startList->type, "child");
       lcp = ch->startList;
       ch->startList = getListNode();
       ch->startList->next = lcp;
       ch->startList->link = startSub;
       strcpy(ch->startList->type, "parent");
     }
       sp = SEARCH_SUB(objName);
       sp1 = SEARCH_SUB(parent);
       lsp1 = sp1->startList;
       sp1->startList = getListNode();
       sp1->startList->next = lsp1;
       sp1->startList->link = sp;
       strcpy(sp1->startList->type, "child");
       lsp = sp->startList;
```

```
        sp->startList = getListNode();
        sp->startList->next = lsp;
        sp->startList->link = sp1;
        strcpy(sp->startList->type, "parent");
}

-------------------------------------------
void putValueMulti()
{
 char *objName,*attName,*type,*value;

 if (num_args() != 4)
  {
   printf("Wrong number of arguments : (putValue ObjName AttName");
   printf(" AttValue AttType)\n");
   return;
  }
  objName = rstring(1);
  attName = rstring(2);
  value = rstring(3);
  type = rstring(4);
  putValueFunctionMulti(objName,attName,value,type);
}

-----------------------------------------
```

- **ACL queries**

```
   iterative-statement(any-cond
                       abstract-expression-description
                       uses : [numOfAtts : & ^ numeral, ?num : & ^ numeral])
          {
              *-statement
                   abstract-expression-description empty;
               actual-assignment-statement
```

```
                    abstract-expression-description
                        uses : [ATTLIST, attFields],
                        defines : [?field : * ^ FIELDS_T];
                *-statement
                    abstract-expression-description empty;
                iterative-statement(any-cond
                                    abstract-expression-description
                                    uses : [?field : * ^ FILEDS_T])
                    {
                        *-statement
                            abstract-expression-description empty;
                        function-call ?printf
                            abstract-expression-description
                            uses : [ATTLIST, Avalue, Aname, AvalueType];
                        *-statement
                            abstract-expression-description empty
                    };
                *-statement
                    abstract-expression-description empty
            }

-------------------------------------------------------
    if-statement(any-cond
                abstract-expression-description
                defines : [?obj : * ^ OBJECT],
                uses : [?SEARCH_CLASS]) then
       {
          if-statement(any-cond
                abstract-expression-description
                defines : [?obj : * ^ OBJECT],
                uses : [?SEARCH_SUB]) then
           {
              if-statement(any-cond
                abstract-expression-description
                defines : [?obj : * ^ OBJECT],
                uses : [?SEARCH_MEMBER]) then
                 {
```

```
                    +-statement
                    abstract-expression-description
                    empty
                }
            }
        }

-------------------------------------------------------------

  for-statement(actual-assignment-statement
                 abstract-expression-description
                 defines : [?i : & ^ numeral];
                any-cond
                 abstract-expression-description
                 uses : [numOfAtts, ?obj : * ^ OBJECT];
                postincrementation-statement
                 abstract-expression-description
                 uses : [?i : & ^ numeral],
                 defines : [?i : & ^ numeral])
      {
          if-statement(any-cond
                 abstract-expression-description
                 uses : [Aname, ATTLIST, ?obj : * ^ OBJECT,
                         ?attName]) then
                 {
                     if-statement(any-cond
                         abstract-expression-description
                         uses : [Multiple, ATTLIST, ?obj : * ^ OBJECT]) then
                       {
                           +-statement
                             abstract-expression-description empty
                       };
                       actual-assignment-statement
                             abstract-expression-description
                             defines : [?field : * ^ FIELDS_T],
                             uses : [?obj : * ^ OBJECT,
                                     ATTLIST, attFields];
```

```
                        *-statement
                              abstract-expression-description empty;
                        any-statement
                              abstract-expression-description
                              uses : [AvalueCast];
                        *-statement
                              abstract-expression-description empty
                    }
        }

-------------------------------------------------------------------------

   if-statement(any-cond
                abstract-expression-description
                uses : [Multiple, ?alist : & ^ ALIST]) then
      {
          *-statement
                abstract-expression-description empty;
          function-call ?putValue
                abstract-expression-description
                 uses : [Avalue, Aname, ?alist : & ^ ALIST, ?obj];
          *-statement
                abstract-expression-description empty
      }
     else
      {
          *-statement
                abstract-expression-description
                empty;
          function-call ?putValue
                abstract-expression-description
                 uses : [Avalue, Aname, ?alist : & ^ ALIST, ?obj];
          *-statement
                abstract-expression-description
                empty
      }

-------------------------------------------------------------------------
```

```
iterative-statement(any-cond
                    abstract-expression-description
                    uses : [?i : & ^ numeral, numOfAtts, ?or : * ^ OBJECT])
    {
        *-statement
         abstract-expression-description
           empty;
        conditional-statement(any-cond
                  abstract-expression-description
                   uses : [?o : * ^ OBJECT, Aname, ATTLIST])
            {
                  *-statement
                   abstract-expression-description
                     empty ;
                  (any-statement
                   abstract-expression-description
                   defines : [notInOrig : & ^ numeral] +
                   any-statement
                   abstract-expression-description
                   uses : [ATTLIST]);
                  *-statement
                   abstract-expression-description
                     empty
             };
         *-statement
           abstract-expression-description
           empty
    }

-------------------------------------------------------------------

  while-statement(any-cond
                  abstract-expression-description
                  uses : [?inchar : & ^ numeral])
    {
       if-statement(any-cond
             abstract-expression-description
                  uses : [?inchar : & ^ numeral]) then
```

```
            {
               any-statement
                 abstract-expression-description
                 uses : [?pos : & ^ numeral]
            };

      (  *-statement
            abstract-expression-description
            empty -

         assignment-statement
            abstract-expression-description
            uses : [?pos : & ^ numeral],
            defines : [?pos : & ^ numeral] -

         *-statement
            abstract-expression-description
            empty  -

         actual-assignment-statement
            abstract-expression-description
            uses : [?pos : & ^ numeral],
            defines : [?inchar : & ^ char] -

         *-statement
            abstract-expression-description
            empty)
      };

    *-statement
      abstract-expression-description
      empty

-------------------------------------------------------------------------

     any-statement
       abstract-expression-description
```

```
          uses : [nextUnique, Aname, ATTLIST, ?obj : * ^ OBJECT, Aname,
                  Multiple, attFields, Avalue, AvalueCast, AvalueType],
          keywords : ["getFieldsNode"]
```

```
  while-statement(any-cond
          abstract-expression-description
          empty)
  {

      function-call fgets
          abstract-expression-description
          uses : [fgets, ?BC_file : & ^ FILE, ?line : & ^ array-type];
      *-statement
          abstract-expression-description
          empty;
      if-statement(any-cond
          abstract-expression-description
            uses : [?BC_file : & ^ FILE, line : & ^ array-type]) then
        {
          +-statement
          abstract-expression-description
          empty
        }
        else
        {
          +-statement
          abstract-expression-description
          empty
        };
      *-statement
          abstract-expression-description
          empty
  }
```

Document Log:

Manuscript Version 1 — March 12, 1994

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX — 12 January 1997

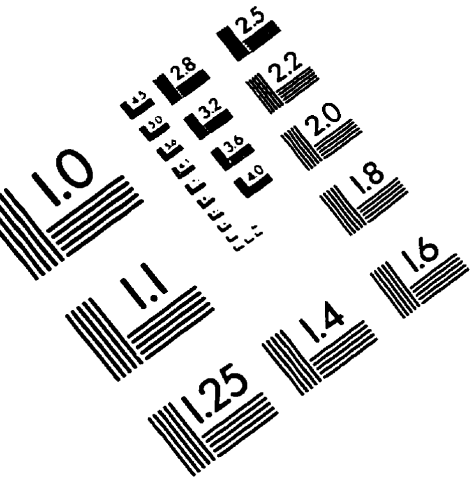
KONSTANTINOS A. KONTOYIANNIS

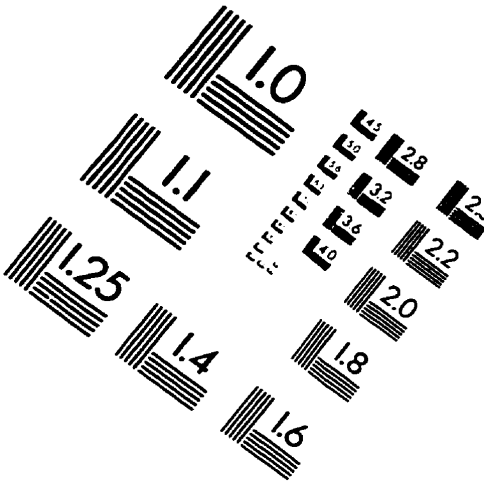
SCHOOL OF COMPUTER SCIENCE, MCGILL UNIVERSITY, 3480 UNIVERSITY ST., MONTRÉAL (QUÉBEC) H3A 2A7, CANADA, *Tel.* : (514) 398-7071

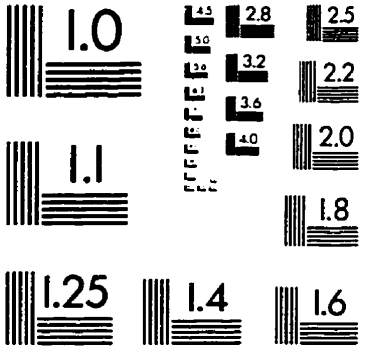
E-mail address: kostas@cs.mcgill.ca

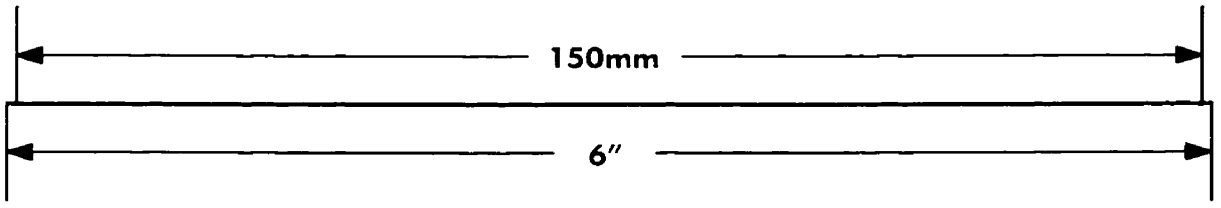
Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX

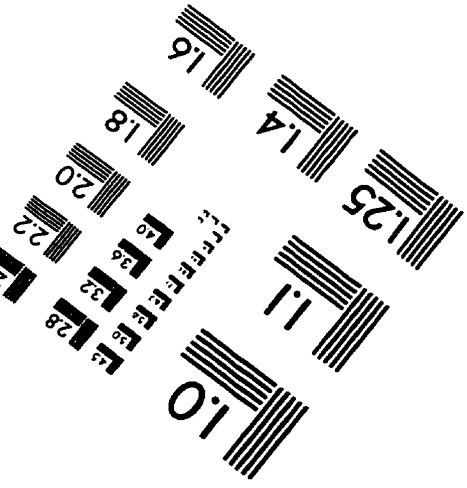
IMAGE EVALUATION
TEST TARGET (QA–3)

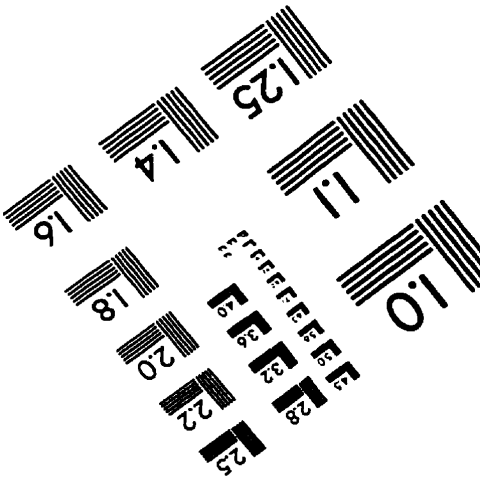












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989