Code Strings: A New Program Plan Recognition Model

By

Michael James Garvin B.Eng.

A Thesis Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements For the Degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University, Ottawa, Ontario

© May 2002

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79702-3

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies and Research
the acceptance of the thesis

**“Code Strings: A New Program Plan
Recognition Model”**

Submitted By

Michael James Garvin B.Eng

In Partial Fulfillment of the Requirements For the Degree of Master of Applied Science

Dr. Trevor Pearce
(Thesis Supervisor)

Dr. Rafik Goubran
(Department Chair, Systems and Computer Engineering)

Department of Systems and Computer Engineering
Faculty of Engineering, Carleton University

May 2002

Abstract

The empirical study of programmers has provided much insight and a great deal of debate. A specific example is the notion of a programming *plan*, i.e. a commonly used sequence of programmatic actions. A non-programming analogy would be a recipe for baking a cake. Much research has focused on supporting or refuting the programming plan concept.

Broad studies of real world projects, to support or refute the idea of programming plans, appear to be lacking. This may be due in part to the absence of an appropriate analysis model, for use by researchers who are not primarily focused on programming. The existing solutions for searching source code for plans are heavily based on regular expression automata, and are colored by the traditional Unix approach of utility scripting. While these approaches work very well for programmers, they are difficult to use and understand by those with less programming background.

This thesis introduces a new graphical model called *Code Strings*, that retains the speed and expressiveness of existing approaches to program plan recognition, and is much easier for non-programmers to use. Both source code and program plans can be converted into Code Strings. Searching for plans is then an exercise in specializing or “instantiating” the program plan Code Strings to match the source Code Strings. A broad survey of twelve real world software projects is conducted to demonstrate the utility of Code Strings, and a new plan instance counting metric or “hits” metric is formulated. Together the survey and metric show that loop based programming plans are the most common.

Acknowledgements

I would like to extend my sincere appreciation to my supervisor, for his unflagging support and patience. His encouragement has been no small part of making my thesis experience worthwhile.

I would like to thank the bottlers of Coca-Cola; this magic elixir has sustained me through many long nights of toil. My teammates and manager also deserve recognition for supporting my thesis efforts, and being patient with me while I manage school and work.

TABLE OF CONTENTS

ABSTRACT.....	III
ACKNOWLEDGEMENTS	IV
TABLE OF CONTENTS	V
LIST OF FIGURES	VII
LIST OF TABLES	VIII
LIST OF ABBREVIATIONS	IX
CHAPTER 1: INTRODUCTION.....	1
1.1 EMPIRICAL STUDY OF PROGRAMMERS.....	1
1.2 PROGRAMMING PLANS.....	2
1.3 EXISTING METHODS FOR DETECTING PROGRAMMING PLANS	6
1.4 ANALYSIS OF CURRENT METHODS	12
1.5 THE CODE STRINGS SOLUTION	13
1.6 THESIS QUESTIONS	14
1.7 MAIN CONTRIBUTIONS	15
1.8 OUTLINE OF THE THESIS	15
CHAPTER 2: THE CODE STRING MODEL	17
2.1 INTERMEDIATE MODELS	17
2.2 CODE STRINGS	18
2.3 CODE STRING CONSTRUCTION.....	20
2.4 NORMALIZATION	22
CHAPTER 3: MATCHING CODE ACTIONS.....	25
3.1 BASIC CODE ACTION MATCHING.....	25
3.2 PATTERN CODE ACTION PARAMETERS	27
3.3 CODE ACTION UNIFICATION	30
CHAPTER 4: MATCHING CODE STRINGS.....	35
4.1 OUTLINE OF MATCHING PROCESS.....	35
4.2 CODE STRING MATCHING	37
4.3 THE MATCHING ALGORITHM.....	39
4.4 A COMPLETE EXAMPLE	43
4.5 BACKTRACKING EXAMPLE.....	50
4.6 DISCUSSION	51
CHAPTER 5: CODE STRING BASED SURVEYING	54
5.1 A SAMPLE SURVEY.....	54
5.2 PROGRAM PLAN SPECIFICATION (ASCII FORMAT)	57

5.3 PROCEDURE	58
5.4 RESULTS	59
5.5 DISCUSSION	64
CHAPTER 6: CONCLUSIONS AND FURTHER WORK.....	66
6.1 CONTRIBUTIONS AND FINDINGS.....	66
6.2 FURTHER WORK	67
BIBLIOGRAPHY	69
APPENDIX A: SURVEY PROGRAM PLANS	71
6.3 ASCII PLAN SPECIFICATIONS FROM SURVEY	71
6.4 ACC1	72
6.5 ACC14	72
6.6 ACC15	72
6.7 ACC2	73
6.8 ACC3	73
6.9 ACC6	73
6.10 ACC7	74
6.11 DG1.....	74
6.12 DID1	74
6.13 DID	74
6.14 DII.....	75
6.15 DII1	75
6.16 LT1.....	76
6.17 SS.....	76
6.18 SL1.....	77
6.19 SL2.....	77
6.20 SL3.....	77
6.21 SL4.....	78
6.22 SWP	78

LIST OF FIGURES

Figure 1 Example Compilation Process (of a C/C++ data swap function).....	17
Figure 2 Code Strings Translation Process (of a C/C++ data swap function).....	19
Figure 3 Example of C/C++ normalization	23
Figure 4 Examples of exact Code Action matching.	26
Figure 5 Example of simple parameter assignment.....	27
Figure 6 Example of greedy parameter assignment.....	28
Figure 7 Example of complex parameter assignment, with inferred simple parameters..	29
Figure 8 Unification Procedure.....	31
Figure 9 Example of Slots used in unification.....	33
Figure 10 Top Level algorithm for matching program plans to source code	35
Figure 11 Example of Code String Matching.....	37
Figure 12 Main matching algorithm (Step 5 of Figure 10).....	39
Figure 13 A source Code String for the implementation of “selection sort”	44
Figure 14 The pattern Code String for “selection sort”	45
Figure 15 Modified source Code String to show backtracking... ..	50
Figure 16 Example of loop jamming	52
Figure 17 Example ASCII coding of a “swap” pattern Code String	57
Figure 18 Analysis and parsing performance across all sample projects	60
Figure 19 Average number of plans found across all projects.....	64

LIST OF TABLES

Table 1 Basic Code Actions.....	20
Table 2 Scope Markers	21
Table 3 Connectors	21
Table 4 Slots for unification attempt #1	46
Table 5 Bindings after unification attempt #1	46
Table 6 Slots for unification attempt #2	47
Table 7 Bindings after unification attempt #2	47
Table 8 Slots for unification attempt #3	47
Table 9 Bindings after unification attempt #3	47
Table 10 Slots for unification attempt #4	48
Table 11 Bindings after unification attempt #4	48
Table 12 Final bindings for all parameters	49
Table 13 Summary of survey.....	59
Table 14 Pattern instances found (patterns 1-5)	60
Table 15 Pattern instances found (patterns 6-10)	61
Table 16 Pattern instances found (patterns 11-15)	61
Table 17 Pattern instances found (patterns 16-19)	61
Table 18 Normalized summary of plan instances found (plans 1-10).....	62
Table 19 Normalized summary of plan instances found (plans 11-19).....	62
Table 20 Average of normalized plan hits across all projects	63

LIST OF ABBREVIATIONS

ESP	Empirical Study of Programmers
GUI	Graphical User Interface
HCI	Human Computer Interaction
CSP	Constraint Satisfaction Problem
PDE	Program Development Environment
AST	Abstract Syntax Tree

Chapter 1:

INTRODUCTION

The study of the psychological and cognitive aspects of programming is not a familiar subject in software engineering or computer science. These disciplines must rely heavily upon software programming to reach their respective goals. With this in mind, finding ways to improve the programming experience, for novice and expert programmer alike, may yield many benefits. One approach to improvement is to look for patterns in the way programmers reach their goals.

This thesis presents a working model of how to look for patterns of programming. This chapter prepares the reader, by introducing the basic ideas behind the study of the act of programming, as well as existing approaches to finding patterns in source code. The Code String model of this thesis is then briefly introduced; a more comprehensive description is left for the succeeding chapter. Finally this chapter concludes with an outline of the goals for this thesis and a short outline of the remaining chapters.

1.1 Empirical Study of Programmers

As computers become more pervasive in our society the need to understand our complex interactions with them has steadily increased. For example, the study of how we use Graphical User Interfaces (GUIs), has grown into a entire field of research, now usually referred to as Human Computer Interaction (HCI). Just as there is a need to understand how a doctor, sales clerk, or child might use a computer in more effective ways, there is a parallel need to understand how a programmer uses a computer.

This field of research is known as the Empirical Study of Programmers (ESP). ESP is complicated by the fact that programmers use computers in multiple modes. Programmers read computer programs to comprehend them. Programmers write computer programs to accomplish a task. When writing a program, programmers may externalize their working memory. Programmers might write part of a computer program, and then temporarily forget it, knowing that the computer will retain the work, and they can return to that piece of work later. Sometimes programmers are engaged in both reading and writing at the same time.

Programming a computer is a complex task. Much research has been conducted, trying to understand how programmers think, how they develop programming skill and what cognitive models can be used to predict more effective ways to program. One cognitive model of programming is that programmers write a program by conceptually breaking down a task into abstract goals, and further into small concrete sequences of source code. Sometimes these sequences of source code are stereotypical; i.e. they are used over and over again. These common sequences are called *programming plans*. These programming plans accomplish parts of the main goal or its sub goals. Programming plans are discussed more in the following section.

1.2 Programming Plans

Programming plans are defined in [Soloway84] as:

Program fragments that represent stereotypic action sequences

An example plan would be a read-process loop. The read-process loop plan is familiar to most programmers. A loop is used to continuously read input from some data source and process that data, until a stopping condition is reached. While programmers can implement this kind of input reading/processing in countless ways, most programmers recognize the read-process loop as a

kind of basic building block of programming. In this way the read-process loop is very stereotypic, and is not closely coupled to any particular programming language.

Plans can also be layered and composed together to form more complex plans. In this way plans can become hierarchical and top down goal driven. Algorithmic plans appear at the top of a task breakdown, tactical decomposed plans in the middle, and fine-grained concrete implementation plans at the bottom. This thesis focuses only on concrete implementation plans.

[Soloway84] also introduces notion of programming discourse:

Rules that specify conventions in programming

These rules can be both informal and formal. An example convention would be that variable names in a function are such that they re-enforce the idea of what a given function accomplishes. Another example comes from languages like C++ and Smalltalk, where programmers often name variables with meaningful phrases, and make the first letter in each part of the phrase upper case, except the first part of the phrase. For example “theFinalResult” might name a variable that holds a final calculation, as opposed to an intermediate one.

Taken together, these two ideas allow expert programmers to very quickly understand and work with programs. By drawing upon a mental library of plans, an expert programmer can quickly deduce the function of a program by scanning its structure, and looking for known plans. Scanning the program is accelerated by being able to infer expectations of behavior, based on known programming conventions. Thus avoiding having to expend large amounts of comprehension energy on every line of source code.

Programming plans offer a model of the programmer's cognitive understanding of a program. Some researchers have tried to take advantage of this by building systems that aid in debugging programs in a more intelligent way, examples include Bug-Doctor [Bellamy90], PROUST [Johnson85] and GAP trees [Spohrer85]. Bug-Doctor and PROUST are essentially systems to automatically analyze programs, infer the plans that are present, and infer from the plans the goals of the program. To aid novice programmers, such systems usually include many dubious variations of standard plans, so that typical novice errors can still be recognized as plans. Bug-doctor uniquely uses a fuzzy logic engine to speed up plan searching. PROUST differs in that it uses a pure knowledge based approach. GAP trees were used to show that buggy novice programs could be broken down from requirements to code in terms of a tree of alternating levels of goals and plans. GAP trees also showed that many novice programmers make mistakes by erroneously merging plans, that some plans are cognitively harder than others, and that some bugs are dependant on other bugs (because the plans they come from are dependent).

Programming plans also appear to be useful in maintaining software, [Gellenbeck91] introduces the idea of "beacons". Beacons in the code can quickly infer programming plans. These beacons are small well known fragments of larger plans. For example a "swap" fragment usually indicates a sort algorithm is present. Expert programmers are so used to recognizing plans this way that they will erroneously identify a plan or miss surface details of the program, when they see a beacon that isn't being used in the standard way. Properly named variables and procedures can also work as beacons, guiding programmers expectations, as they comprehend a program.

Programming plans have not been broadly accepted as a standard cognitive model of programming. Even though systems like PROUST and Bug-Doctor show the utility of plans, some researchers have questioned the cognitive validity of plans. [Soloway84] suggests that

plans are universal and generalize to all languages, and that experts rely much more on plans than novices. [Soloway84] showed that when presented with unplan-like programs, experts suffered a performance penalty, whereas novices did not. Other researchers have found plans to be a notational artifact of converting the mental model into an implementation model [Gilmore88] [Bellamy90]. [Davies90] takes the viewpoint that plans are sometimes used as the internal mental structure, sometimes a product of implementation, and sometimes an after effect of teaching or design experience.

Some researchers have noted that programmers don't always implement a given task in a top-down breadth first way, rather they jump around, making use of programming "opportunities" as they arise, this result is cited as evidence that programming plans are not used. However, [Ball95] argues that opportunistic programming is in fact a depth first strategy or a mix of depth first and breadth first, and that programmers are really showing strategic prioritization in how they implement their plans.

Programming plans are intended to be very general and not specific to any one programming methodology. For example Programming plans are not "design patterns" as put defined by the C++ programming community. While programming plans can be used to find general programming plans within Object Oriented (OO) source code, they are not specifically intended to find design patterns or other higher level programming plans. Programming plans focus on the concrete level of a program, where as design patterns are more abstract and combine both control flow and dataflow between objects and functions/methods. Programming plans could be extended to work with design patterns, but would require much more data modeling.

In a similar manner programming plans are not intended model higher level patterns such as real-time or concurrency patterns.

1.3 Existing Methods For Detecting Programming Plans

Current approaches to plan recognition fall into four categories:

- Purely lexical/AST based analysis
- Graph parsing
- Knowledge based analysis
- Constraint Satisfaction

In addition to basic detection of programming plans, some research has involved the development of Program Development Environments (PDEs) that support programming plans from design through to implementation. The motivation is to provide a seamless programming environment that can trace requirements through to code, and reduce the low level programming burden on designers. The most well known examples of such PDEs are The Universe PDE [Parker87], and The Programmer's Apprentice [Rich88][Rich90]. The Universe PDE does not appear to offer program plan detection, and focuses solely on forward engineering. The Programmer's Apprentice though, supports program plan detection via graph parsing (as described below), and forward engineering.

Purely Lexical/AST Based Analysis: For a computer to understand a program, which is written in human readable text, a process of translation must occur. The human readable instructions or "program" are converted to a machine-readable structure. This intermediate data structure is then converted into an optimal stream of machine instructions that can be understood by the computer the program is to run on. The term of art for this conversion process is "compiling a program". The computer program that does the compiling is called a "compiler". When a

compiler reads a program, it must first understand what exact words the programmer has entered, despite comments, extra spaces between the words, or other special processing. This basic act of reading the words or “tokens” in a program is called lexical analysis. Once the tokens of a program are understood, they can be connected into a higher-level grammar, to give them proper meaning.

The compiler will usually construct a data structure to hold the grammar of a program. This data structure is usually called an Abstract Syntax Tree (AST). Once the AST is constructed, it will be used for various forms of analysis so that the compiler can produce an optimal set of instructions for the computer to run. Constructing an AST is much more complicated and resource intensive than lexical analysis.

For lightweight lexical analysis, the most well known approach is LSME [Murphy96]. LSME allows the user to define regular expression token machines. A formal definition of regular expressions:

Regular languages are the smallest class of languages, which contain all finite languages and are closed with respect to union concatenation and Kleene closure (repetition). If I is an alphabet and R is a regular expression over I , then R must conform to the rules:

$R = \epsilon$ (empty expression)

R belongs to I

$R = R_1 \cup R_2$ (set union, R_1 and R_2 are regular expressions)

$R = R_1R_2$ (concatenation)

$R = R_1^*$ (repetition or Kleene closure)

For example A^*B is a regular expression that denotes the set of strings formed by a sequence (also empty) of A symbols followed by a B symbol. Regular languages can be recognized (matched) by finite automata and are generated by regular grammars.

More informally, a regular expression is a definition of a language with the letters/ words drawn from a very finite set. In addition a regular expression defines some simple rules of combination. For example a trivial language definition might be "a". This definition says that the only valid instance of the language is when the token "a" is present. A slightly more complex language would be "a b|c". In this definition a valid instance of the language is present when a sequence of tokens starts with "a" and is then followed by either "b" or "c". The | character is used as short hand for indicating an alternative at that point in the grammar of the language.

A regular expression machine is a computer program, usually generated automatically by another program, from a regular expression. The regular expression machine can then be used to recognize a language or a pattern in a program by processing the program as a sequence of tokens, from the very first token, to the last token. When a pattern is found in the program, the regular expression machine will indicate the position. The lexical analysis of a program is an advanced form of a regular expression machine.

LSME can convert regular expressions into scanners (a kind of regular expression machine). Scanners act like an assembly line machine; they take in several parts (tokens), as the program source is pushed through the machine, and when a specific pattern is found, they indicate a match. LSME allows multiple concurrent scanners to be run on a body of source code. Scanners can, in turn process the results of scanners. This allows for inter-file/inter-procedural analysis.

While this lexical approach is good at finding specific pieces of code, and doesn't require an AST, it is not particularly well suited to finding programming plans. The lack of syntactical matching (i.e. no knowledge of the AST), means plans must be specified exactly, token for token, thus excluding many matches to a more abstract plan, and only allows for that exact instance.

A more powerful approach to detecting program plans is to work with the AST for a program. The Scruple system [Paul94] was the first to exemplify this approach. Regular expressions were based on elements of the AST rather than tokens alone. The regular expression syntax was difficult to work with though, and a complex implementation was required. The matching machine had to be "walked" over the AST. Scruple worked on the AST for a whole program at once, thus limiting the scale of a program it could work with.

While able to handle very rich queries, Scruple only worked with the concrete elements of the AST, so when searching for a loop, patterns would have to be provided for each kind of loop. In a C like language this means having patterns to handle `for`, `while` and `do` loops, rather than the abstract element "loop". The current survey software implementing the Code Strings based matching engine is limited in the same way.

A more advanced version of the AST analysis approach called TAWK [Griswold96], implemented a more granular approach, allowing the AST for just a file or procedure to be examined at once, mitigating the scale issue. TAWK is an extended model of the AWK scripting language. The essential idea behind AWK is that a sequence of text can have some number of string matching regular expressions applied to it line by line. When a match is found a corresponding action is performed, i.e. a script of user commands. In TAWK the regular expressions are based on syntactical elements of the AST, and are used for searching for the plan

patterns. C programs instead of scripted commands implement the match actions. The C programs must be compiled and dynamically linked with the TAWK system.

TAWK can work with both concrete and abstract elements of the AST, and attempts to provide an analysis that is based on what the source code looks like to a programmer, rather than its post processed or compiler ready form. TAWK can provide very powerful queries for programmers, but non programmers would find TAWK intimidating, if only for the implied understanding of an AWK like model, and coding of actions in C.

Graph Parsing: This approach is best exemplified by the Programmer's Apprentice project [Rich88][Rich90]. The approach taken was to convert both program plan patterns and source code, into a formalism called The Plan Calculus. Essentially The Plan Calculus is a graph representation that allows for data, control flow and abstract data types to be represented in a graph. Because both program plan patterns and source code can be converted into The Plan Calculus, searching for plans becomes a graph parsing with constraints problem. While workable, this system is subject to combinatorial explosion, and fails to scale to even moderately sized systems.

Another well-known graph parsing approach is that of [Kozaczynski92]. Here a concept hierarchy is used in attempt to build design models of a given program. At the bottom of the concept hierarchy for a program are concrete plans; these plans are found by matching sub trees of the AST for a program, to AST versions of program plans. This approach is very intensive, high levels of concept recognition require extensive data/control flow analysis to find appropriate constraints to match against.

Knowledge Based Analysis: This approach is typified by a system such as PAT [Harandi88]. This approach converts the given program into a set of facts. The facts are stored in a database. An inference engine is then used to develop explanations or proofs of what plans exist in the program. Once concrete instances of plans are found, the inference engine can infer high level concepts. Introducing new program plans is difficult with this approach, as it requires extensive knowledge engineering.

A more recent example of this approach is the KARE system [Palthe97]. KARE uses a concept hierarchy, and graph parsing to find the basic concrete program plans in a given program AST. KARE provides a very sophisticated knowledge model to aid in inferring higher-level concepts. Adding new programming plans to KARE is a difficult task, as with PAT much knowledge engineering is required. In addition KARE focuses on attaining optimal search times, by having the user select only small portions of code to search.

Constraint Satisfaction: This approach to finding program plans uses a generalized model. The elements of an AST are considered to be values in the search domain, the program plan to search for is the constraint, and finding ways to parameterize the given plan to match it to a location in the code, is a satisfaction of the constraint. In this way program plan recognition is viewed as a Constraint Satisfaction Problem (CSP). It can also be viewed as specializing or “instantiating” the program plan, to match some sub-tree of the AST (a location in the source code grammar).

This approach is championed by [Quilici93] [Quilici94], and uses a program AST to find concrete elements of a program. Unfortunately, this approach is very prone to combinatorial explosion. Much effort has been placed into optimizing this approach [Woods95], [Woods96], [Quilici97], however the approach still does not appear to scale past 10-20 thousand lines of code. This model provides a concise unifying framework for understanding plan recognition.

1.4 Analysis of Current Methods

All of the current research has focused on studying small groups of programmers, on “toy” programs. Using larger groups or amounts of code becomes impractical because of the detailed work involved in the protocol of such an experiment. Even if such a study were done, its value would still be related to what was observed “in the lab”. None of the existing approaches has been used in a broad survey of real world code. Some have only been tested on “cooked” code.

A system like TAWK can be very useful to a knowledgeable programmer, but none of these approaches has provided a graphical model that is easy to use by non-programmers, and requires very little learning curve. All of these approaches require the AST for a program, this can add to the brittleness of the tools using this approach, and adds complexity and performance constraints. Of these systems TAWK can tolerate an incomplete AST, but like Scruple is greatly complicated by having to walk a regular expression machine over the AST.

TAWK is interesting though in its suggestion that it can be more valuable to analyze what a program looks like to a programmer, rather than to the compiler. For example in parameterized languages (such as C++), a template may not yet be instantiated, and an AST may not be available, yet if the source code could be analyzed “as is” with entirely incomplete types and very little data structure, then plan recognition might still proceed.

Of all the existing methods the approaches taken by Scruple and TAWK appear to be best suited for plan recognition in large amounts of software. However the approach taken by these tools requires a complex implementation. In addition the user is expected to be very familiar with

regular expression based languages that derive from AWK (a Unix scripting language). An approach that is simpler to implement and requires less technical knowledge by the user would be more desirable.

1.5 The Code Strings Solution

This thesis proposes a new approach to recognizing patterns in programs. A visual variation of regular expression machines is defined and implemented. This model of pattern recognition in programs is called "Code Strings". The Code Strings model requires no knowledge of regular expression languages, nor does it require a complex implementation. The Code Strings model retains some of the expressiveness of AST/graph parsing approaches, but is simpler to implement and use, like the lightweight lexical approaches. The Code Strings model does not rely on knowledge based analysis, but does use constraint satisfaction to aid in the pattern matching processing.

In order to remove the doubts associated with small "in the lab" experiments. Real world software systems, both small and large need to be surveyed. This kind of survey data can be used for supporting or refuting the utility of programming plans. Rather than study programmers, study the code they produce and study a lot of it. If a survey could be taken of a very large and broad amount of source code (say millions of lines of code), from a variety of software projects ongoing today, then that survey might provide a physical fingerprint of plan usage. If plan usage is common enough, and obvious enough in real software, then an empirical basis for the utility of programming plans will have been found.

The proposed Code Strings model seeks to advance program plan recognition, by providing a purpose built technique. Both program plans and source code can be converted into Code Strings. These Code Strings can then be used for matching the pattern strings (program plans) to

the source code strings. The Code Strings model aims to provide a very simpler means for non-programmers to specify program plan patterns. The Code Strings model also seeks to overcome the scale issues found in many of the current approaches, and demonstrate that it is possible to survey large amounts of real source code, and provide a very strong empirical basis for programming plans.

The Code Strings model is intended to be general and not specific to any particular programming or design methodology. In addition Code Strings are not designed to model data flow in great detail. As such Code Strings cannot directly model higher-level notions of source code patterns like "design patterns" from the OO programming community. However Code Strings could be extended to incorporate more dataflow modeling, and then applied to these higher level notions of source code patterns.

In a similar manner Code Strings are not immediately intended to be applied to real-time or concurrency patterns.

1.6 Thesis Questions

The primary question of this thesis is:

Can a model of program plan recognition be developed that requires less technical knowledge and is easier to implement than existing methods, without entirely sacrificing performance or expressiveness?

The secondary question of this thesis is:

*Can such a model be used to survey a wide range of code and provide an empirical basis
for supporting or refuting the notion of programming plans?*

1.7 Main Contributions

This thesis provides a strong yes to the primary question, and implements software to conduct surveys of C/C++ software. This model could be adapted to other languages as well. The secondary question is open ended but partially answered by way of a broad survey, involving millions of lines of code, from software projects in various application areas. No other such survey is known to exist. The survey was conducted using the software implemented for this thesis work. However, further surveying could build up a more complete answer. In addition a new metric for counting plan instances or “hits” is formulated. The software developed for this research, and the source code surveyed, are publicly available at <http://www.sce.carleton.ca/~garvin/codestrings>.

While a graphical language was developed for specifying Code Strings, it was not a central focus of this thesis work. When developing the Code Strings model, a graphical view of a Code String became a useful tool for discussing Code Strings. This graphical view of Code Strings has not yet been developed into a full language, nor has a GUI for manipulating such a graphical language been developed. Developing a strong version of this graphical language and its well-formedness rules has been left for future work.

1.8 Outline of the Thesis

Chapter 2 introduces the Code String model. Code Actions are introduced as the basic unit in constructing Code Strings. Some simple examples are used to demonstrate how an actual program or a program plan can be represented as a Code String.

Chapter 3 develops the idea of Code Action matching. Matching of Code Actions is the cornerstone of Code String matching. Constraints, and various kinds of pattern Code Action parameters are introduced. The chapter then builds on these ideas to show how unification of Code Actions can be accomplished. With the ability to specialize a pattern Code Action to exactly match a source Code Action (Code Action unification), the algorithm for matching strings can be developed.

Chapter 4 builds on Chapter 3 by using the Code Action unification process to match Code Actions, and adds a higher level of matching of Code Strings. With the matching algorithm in hand, this chapter concludes with a fully worked example.

Chapter 5 demonstrates the utility of the Code Strings model by surveying a wide selection of real world code, and provides a partial answer to the secondary question of this thesis. Chapter 3 and 4 taken together, answer the primary question of this thesis.

Chapter 6 concludes the thesis by summarizing the thesis contributions and discussing future directions of this thesis work.

Chapter 2:

The Code String Model

This chapter introduces the ideas behind the Code String model. The process of converting software (source code) into Code Strings is discussed in detail. Codes Actions which form the basic units of Code Strings are also introduced. Later chapters will build on these definitions to develop the matching algorithm for finding programming plans in source code.

2.1 Intermediate Models

When analyzing a program in some way, a translation process usually precedes analysis.

Figure 1 Example Compilation Process (of a C/C++ data swap function)

For example when the source code for a program is compiled, to produce a binary program that can run on a computer, there are several steps involved. An example compilation process is shown in Figure 1. In this example, the intermediate model is an AST. The AST provides a clear meaning of the program, and can be analyzed to produce an optimal program to run on the computer.

2.2 Code Strings

The part of the compiler that generates the AST is usually referred to as the parser, since it parses the lexical tokens to generate the AST. However, a parser for a given language could be reworked to produce an alternative intermediate model. The alternative model proposed by this thesis is the Code String. A simple example is shown in Figure 2.

A Code String is a weaker, simplified grammar that essentially captures the control flow of a function/method. Code Strings are an intentionally relaxed grammar to allow for matching in source code that may not be complete or in form that can be properly compiled. Being a relaxed grammar also allows for much simpler specification of program plan patterns. A simpler grammar also allows for a simpler and faster implementation.

The drawback of Code Strings is that they can be too specific. When a slightly different version of a program plan is needed, an entirely new program plan pattern must be specified. Future work should include extending the Code Strings model to allow for more variation. In addition the Code Strings model can identify false positives. These are cases where a plan instances were identified, but the given plan is not actually being used in that instance. This happens when program plan patterns are made too general, in an effort to match more variations in the source code syntax.

Since the current parser does not expand macro's or templates, it is also possible that match could be falsely identified simple because a macro or template has not been fully expanded. On whole though it is expected that a model based on what is immediately apparent in the source code is more valuable.

Figure 2 Code Strings Translation Process (of a C/C++ data swap function)

As shown in Figure 2, the source code can be converted into a Code String, rather than an AST. A Code String differs from an AST in that it does not attempt to parse the tokens of the program into a grammar, and thus construct the meaning of the program. Instead Code Strings only attempt to represent the control flow and major actions performed. As was shown in this example, the control flow is quite simple; a linear chain of actions. Here the actions are a declaration, followed by several assignment expressions. The actions are more formally referred to as *Code Actions*. How to form Code Strings from Code Actions is described in the following section.

2.3 Code String Construction

When translating source code into a Code String, each action encountered is converted into a certain kind of Code Action. For example, the start of a loop will be converted into an "Iterate" action, most expressions can be converted into an "Expression" action, and control flow actions like breaking out of a loop, or returning from a function, are converted into "Jump" actions. The basic actions are shown in Table 1.

Table 1 Basic Code Actions

Icon	Code Action	Description
Iterate	Loop header, for/while/do etc. Normally used with Scope Head/End.
Jump	Branching primitive break/return/goto/if/else/switch/continue
Expression	A single statement, assignment or execution of an expression
Declaration	A variable declaration (any initialization expression is consumed with the declaration)

Code Strings model the control flow of a function or method. Since there is a definite scope to a function or method, special markers are used to start a Code String and end a Code String. These special Code Actions are called the Scope Head and Scope End respectively, and are shown in Table 2. Scope actions occur not just at the outer scope of a function or method, but also wrap

any nested scope. For example all loops and conditional statements have a scope. Sometimes these scopes are not made explicit in the source code. This issue is dealt with in the Normalization section (2.4).

Table 2 Scope Markers

Icon	Code Action	Description
Scope Head	The beginning of a nested scope (i.e. '{')
Scope End	The end of a nested scope (i.e. '}')

In order to construct a Code String from Code Actions, the Code Actions must be linked together. There are two ways to link Code Actions together; the first way is with a fixed length connector. Such connectors imply that one Code Action immediately follows the previous one without any other Code Actions appearing between them. A variation of this is a branching connector that is used to split a scope into the existing scope and a new nested scope. This branch connector is used to hang sub-strings off of parent strings. Sub-strings model the nesting of scopes. All sub-strings begin and end with scope actions. These connectors and a special "spring" connector are shown in Table 3

Table 3 Connectors

Icon	Code Action	Description
Spring	This is a wild card, it models 0..n arbitrary Code Actions.
Fixed Connector	Chains two Code Actions together, implies no other actions can occur between the two that are connected.
Scope Branch	Allows the splitting of scopes. The RHS will be immediately followed by a Scope Head for the nested scope

Spring connectors are used in Code Strings that model a pattern of source code, as opposed to actual source code. When matching a pattern Code String to a source Code String, it is sometimes necessary to allow several sequential Code Actions from the source Code String to be

ignored. Code Strings must have a special code action that can “consume” an arbitrary number of Code Actions from the source Code String. The spring Code Action serves as a wild card.

As was shown in Figure 2, a single scope function is translated into a source Code String. Each of the major actions in the function is modeled with a basic Code Action. The beginning and ending of the function are marked with scope actions. Since this is not a pattern Code String, all actions are “bolted” together with fixed length connectors. Since there are no nested scopes, there is no need for any branching connectors.

Most basic Code Actions involve not just an action, such as jump or expression, but also operations and operands. In Figure 2 the first action involves an assignment operation (=) with two operands “tmp” and “a”.

2.4 Normalization

During the parsing of source code, normalization is performed. Normalization is the act of transforming the Code String of the source code, so that it can be compared more consistently to pattern Code Strings. In general normalization is the act of removing syntactic variation from two Code Strings that are semantically equivalent. Currently this Code Strings model only normalizes scope differences, as described below.

In some languages, it is possible to imply a nested scope, without actually specifying the boundaries of the nested scope. For example, a C/C++ conditional statement has a compound statement (multiple statements in an explicit scope), or it can be followed by just a single expression. The conditional statement has scope over both situations, but in the latter there is no explicit indicator of the scope.

Implicit scope becomes an issue, because some programmers may include the scope indication, and some may not. So, even though two programs might be coded to do the same thing, the difference in scope indications can produce different strings. An example of two such functionally similar, but syntactically different pieces of code is shown in Figure 3.

Figure 3 Example of C/C++ normalization

Even though the scope indications of each piece of code are different, they both have the same control flow. The Code String for the first piece of code is modified, so that it includes the missing scope indicators. This insertion of synthetic scope markers, where necessary, is called normalization.

Currently the Code Strings model normalizes the scope indicators for both conditionals and loops. No normalization is attempted on expressions or other parts of the language. For example

expressions like $i++$, $i = i + 1$ and $i += 1$ are not normalized to just $i++$. Such expression based normalization could be done and would aid in matching more instances of programming plans. However this extension has been left for future work.

Chapter 3:

Matching Code Actions

This chapter focuses on developing the ideas necessary for matching Code Actions. Before discussing how Code Strings as a whole can be matched, the mechanics of matching individual actions must be presented. The basic rules of equivalence are introduced, followed by the ideas of constraints and various kinds of parameters. Finally these ideas are brought together to implement Code Action unification, which is the cornerstone of matching Code Actions, and hence matching Code Strings.

3.1 Basic Code Action Matching

Codes Strings are composed of Code Actions. Code Actions are tied together in sequential order, potentially with nested sub-strings. To match one Code String to another Code String, the Code Actions of each Code String must be compared. The ordering of Code Actions in each Code String must also correspond. Code Action matching is discussed here while “string” level matching is discussed in greater detail in the next chapter.

Any two Code Actions are considered roughly equal if they are of the same kind. A “jump” Code Action can only be equal to another Jump Code Action. For an exact match their operators/operands must be equal as well. Connectors and springs are never matched. The jump, iterate and expression Code Actions are the only actions that can be matched.

Figure 4 Examples of exact Code Action matching.

As shown in Figure 4 the operator ($+=$) of the expression is being used as a constraint in matching. In addition the operands are used to constrain matching, even the ordering of the operands is important, since in most languages the left to right ordering effects the result of the operation. In a) the action type, operation and operands are all the same, so the two actions are equivalent. In b) the operations are different, so there is no match. In c) the operands are different so there is no match. In d) the ordering of the operands is different, so there is no match.

In this example there is only one operator in the expression, however a given Code Action may have multiple operators and associated operands. For example, in C/C++ a `for` loop will nominally have 3 operators (for initialization, loop end and loop variable increment). When matching Code Actions with multiple operators, the operators are matched in the left to right order, as they are found in the source code.

Note that expressions in Code Actions are sequences of tokens. Code Action expressions cannot be evaluated in a mathematical sense. This introduces a limitation to Code Action matching. The expressions must be matched in a very concrete way. Expressions are not normalized.

3.2 Pattern Code Action Parameters

To allow for matching Code Strings with syntactic variation, but equivalent semantic meaning, any two Code Actions need to be compared in a way that allows the operands to be made equal "on the fly" or unified. If one Code Action is a pattern action, and the other Code Action is a source action, then the pattern action, can have its operands generalized to parameters. The operators and their operands together then become "slots" (as shown in Figure 9) that can be filled in as needed to create a match. The pattern action parameters can be bound to operand names as needed. This allows the pattern action to initially be a general pattern action, it can then be specialized via parameter assignment, to exactly match or unify with the source action. Examples of Code Action unification are shown in Figure 5 and Figure 6. The full unification algorithm used in matching Code Actions, is discussed at the end of this chapter.

Figure 5 Example of simple parameter assignment

As shown in Figure 5 a) the two Code Actions are not equal, but the pattern Code Action, has the parameter "\$1" in the place where the two Code Actions differ. Parameters are given special

names that begin with a dollar sign, and end with a number. The number is used to distinguish parameters. In this case there is only one parameter, namely \$1. Since there is only one token difference in the operand that is different, the parameter only needs to be assigned a single token, to unify the two Code Actions. This unification is shown in b). Since only one token is required for unification, this kind of parameter is called a simple parameter. Simple parameters can only unify with or "consume" a single token. If the left hand side operand had differed by more than one token, then a simple parameter would not have been sufficient to unify the two Code Actions.

Figure 6 Example of greedy parameter assignment

As shown in Figure 6 operands can differ by more than one token. The two Code Actions can still be unified, but a different kind of parameter is needed. This kind of parameter can consume any and all tokens that make up an operand (once side of an operator). Because of this behavior, this kind of parameter is called "greedy" instead of simple. The parameter name is distinguished as greedy, with the addition of a plus sign to its name, as shown in a). The multi-token unification is shown in b).

Only the operators supplied in the pattern action will be matched against the source actions. An operand is always represented by a sequence of tokens. Tokens are non-white space lexical tokens.

In some situations, the value for a parameter must be inferred from the context in which it occurs. For these situations a third kind of parameter is needed, that is between greedy and simple, it consumes multiple tokens, but only by matching the tokens and simple parameters present in its definition. This kind of parameter is called a complex parameter. A complex parameter can only depend on tokens and simple parameters. A complex parameter may not have greedy parameters within its definition. Future work may extend the Code Strings model to allow for greedy parameters inside complex parameters. An example is shown in Figure 7.

Figure 7 Example of complex parameter assignment, with inferred simple parameters

The definition of \$3 needs to find two simple parameters within the context of two square brackets. This context is found, and the appropriate substitutions are made:

$\$1 \rightarrow a, \$2 \rightarrow c$

Now the definition of \$3 becomes:

$\$3 \Rightarrow a[c]$

This multiple token parameter can now be matched directly to the left hand side operand in the other Code Action.

As shown in Figure 7 a) the parameters \$1 and \$2 are simple parameters, but are not yet known. However, given the context (the two square bracket tokens), we can infer the single token values for \$1 and \$2. With these values substituted, we can match the complex parameter \$3 as shown in b).

In this example the parameter \$3 is not really needed to unify the two Code Actions. However in general having \$3 in the plan pattern is useful because it may need to be instantiated in other Code Actions to help match the entire program plan. Allowing simple parameters like \$1 and \$2 to be aggregated up into complex parameters like \$3 allows for better context driven pattern matching, better “locks” be achieved when looking for program plan instances.

3.3 Code Action Unification

As was shown in the previous examples, unification is a critical part of Code Action matching. For a match to occur between a source Code Action and a pattern Code Action, the latter must be an exact match, or the latter must have parameters that can be “bound” or assigned the values of appropriate operands in the former action, thus unifying the two Code Actions and making them equivalent.

When attempting to unify a pattern Code Action to a source Code Action, there are three cases to consider:

- 1) The operands of all operators are the same in each code action. All of the tokens that make up each Code Action are an exact match.
- 2) The operands of some operator’s match, but other do not, and there is no way to unify, hence no possible match.

- 3) The operands of some operator's match, and where there are differences, the pattern action has a parameter.

The first case is the trivial exact match, and is not expected to occur frequently. The second scenario is expected to happen very frequently, as most of the source code being searched, will not contain the pattern being searched for.

The third case is the core of matching, given that two Code Actions have similar operators, and the pattern action has free parameters to be assigned, we need only assign the operands of the source Code Action, to the parameters of the pattern Code Action, in order to unify the two Code Actions and create a match. These scenarios are also illustrated in Figure 4 - Figure 7. The outline of the unification procedure is shown in Figure 8.

Figure 8 Unification Procedure

1. FOR each pattern slot & source slot DO
2. Make a working copy of the current pattern slot, so that if unification fails, we go not have to worry about "undoing" things in the main match algorithm.
3. Find the next plausible source slot.
4. IF no appropriate source slot THEN
5. We couldn't find a plausible source slot to try unifying with, this action cannot possibly unify. Abort.
6. RETURN FALSE
7. ENDIF
8. Before attempting to make new bindings, we substitute any known bindings for both simple and greedy/complex parameters.

9. Look for new simple parameter bindings.
10. Substitute any new simple parameters.
11. IF pattern slot has greedy/complex parameters THEN
12. Look for greedy/complex parameters
13. ENDIF
14. Substitute any new greedy/complex parameters.
15. IF constraints not satisfied THEN
16. Report unsuccessful unification
17. RETURN FALSE
18. ENDIF
19. ENDFOR
20. Report successful unification
21. RETURN TRUE

As shown in Figure 8 step 1, unification proceeds left to right, operator-by-operator and operand-by-operand. Each operator is also called a “slot”, since the operators are where parameters can be substituted or bound during unification. Constraint matching is one way; whatever appears in the pattern must appear in the source string. The contents of the source string do not necessarily need to occur in the pattern string. If the source string has extra operators/operands then they will just be ignored. Note that greedy parameters never consume tokens beyond the scope of a slot.

In steps 2 and 3, we advance the pattern slots to the next one, and we advance the source slot to the next one that looks like it could be used for unification. The operator ordering in the pattern is left to right, and the pattern operators are only used to constrain the matching. Before starting

unification proper, we first substitute any known parameters if parameters are present in the pattern slot. Step 8 is simply a pasting of the previously bound token sequences.

Figure 9 Example of Slots used in unification

For example in Figure 9, the first pattern slot has an free parameter (\$3). The first slot can be unified by binding "i" to parameter \$3, the remaining operator ("=") and operand ("0") match token for token. Note that \$3 is a simple parameter. The second slot is unified by first substituting for parameter \$3, and introducing a new binding for parameter \$4 ("5"). Finally the third slot is unified simply by substituting the known binding for parameter \$3. To unify the third slot, a greedy parameter could also have been used. For example a new parameter \$5+ could have been used. Because \$5 is greedy, it would have consumed the entire slot "i++".

Having already replaced known bindings, we look for new bindings to the remaining free simple parameters in step 9. We don't look for new greedy/complex parameters yet, since they may depend on simple parameters. If any new simple parameters are generated, we substitute them immediately in the pattern slot, again to reduce the act of unification to a simple token comparison. Greedy and complex parameters are illustrated in Figure 6 and Figure 7.

If the pattern slot has greedy/complex parameters then during step 12, both complex and greedy parameters will be searched out. Greedy parameters can consume entire expressions (i.e. all of an operand). Any new parameter bindings that are generated are immediately substituted. If there are still free parameters in the pattern slots, or the operands of each slot do not yet match, then we were unable to unify the slot. Failure to unify a slot implies that the whole action cannot be unified. If all slots unify then we have satisfied the constraints of the pattern and we can return successfully.

Chapter 4:

Matching Code Strings

With the Code Action concepts thoroughly introduced, this chapter extends the notion of Code Action matching to Code String matching. The outline of the main matching process is described, and then expanded upon. Finally, the chapter concludes with a fully worked example of how to apply the algorithm, including backtracking.

4.1 Outline of Matching Process

Code Strings model the control flow of a function or method. At this level of granularity, program plan pattern Code Strings must represent some part of or all of a function or method. Hence program plan pattern Code Strings must be matched to each and every function or method in the body of source code to be searched. A simple but brute force approach to matching at this level of granularity is shown in Figure 10.

Figure 10 Top Level algorithm for matching program plans to source code

1. FOR each source file DO
2. FOR each source Code Action string in the current source file DO
3. FOR each pattern file DO
4. FOR each pattern Code Action string in pattern file DO
5. match the pattern Code String with the source Code String
6. IF matched DO
7. report the results
8. ENDIF

9. ENDFOR
10. ENDFOR
11. ENDFOR
12. ENDFOR

As shown in Figure 10 each source file must be broken into several Code Action strings, one for each function or method. These strings are called the “source strings”. Correspondingly, the program plans are specified as functions in the program plan pattern files (there is no GUI for specification as yet). These too must also be broken into a set of Code Action strings, called the “pattern strings”. The match algorithm is then a process of trying to match pattern strings to source strings.

Once a program plan is matched in a function/method, the instance is counted and the search algorithm stops looking for further instances that occur later in the function/method. Future work should include updating the search algorithm to count all instances of a plan in a given function/method. Multiple program plans can be found within the same function/method, however the program plans must each be specified on their own. The current specification method does not allow for nested or interwoven plan specifications.

Since matching Code Strings involves matching their Code Actions, unification of Code Actions is a key part of matching. Code Action unification casts string matching as a process of specialization. This specialization or “instantiation” is accomplished by parameter substitution and/or spring stretching, as was detailed in the previous chapter. To match a pattern string to a source string, the pattern is specialized action by action, until it becomes an exact match to the source string, at some position in the source string.

Pattern strings are obtained in the same fashion as source strings. The program plans are written as C/C++ functions/methods, the only proviso is that the plan pattern files may contain some extra syntactical elements to indicate the position of springs or the use of parameters. The extended syntax is discussed in greater detail in Chapter 5. Beyond the extended syntax though, program plan pattern files are parsed in the same way as source code files and converted into Code Action strings. Future work may see the addition of a GUI to allow program plan patterns to be specified in an even simpler manner. The details of matching one string to another are presented in the succeeding section.

4.2 Code String Matching

Step 5 of Figure 10 involves the matching of a pattern Code String to a source Code String. For example if an assignment followed by a `for` loop is the pattern to look for, then that pattern can be formed by an expression Code Action, followed by an iteration Code Action. An example of this pattern is shown in Figure 11. The order in which Code Actions are matched is indicated with the circled numbers. Two Code Strings are considered a match if all of their Code Actions match, as was detailed the previous chapter.

Figure 11 Example of Code String Matching

Pattern strings can only stretch where there are springs. In Figure 11, a spring is used between the assignment action and the loop action. This spring allows for stretch between the two actions when a source string contains other kinds of actions that don't help the match. The Code Strings model allows spring to have a fixed length, thus constraining how far they can be stretched. In the current survey work this kind of constraint was found to be unnecessary, and all springs can be stretched as far as needed. Springs do not have to appear the end of a Code String (or a sub-string). Springs may be used at the end of a string to force a match to the end of scope, however a match is considered "found" as soon as the last Non-spring Code Action in the pattern is found.

In order to allow for nested scopes, Code Strings are recursively defined, at any point in a Code String, a sub-string can be started. This model requires that when matching strings, as soon as a sub-string is found in the pattern string, then that pattern must be matched, in a corresponding nested scope in the source string. Sub-string matching works in the same way as matching for a normal "top level" string, the only requirement is that the sub-string be completely matched, before the upper level string matching can continue.

A pattern sub-string can be matched inside a corresponding sub-string of the source string, but could also be matched inside any further nested scope if there was one. In this way sub-string matching can "float" inwards to inner scopes where needed.

For the example in Figure 11, matching would move down the main string, then into the scope of the `for` loop, and then resume at the outer most level, once the entire scope of the `for` loop had been matched. In the pattern Code String, the split point between the loop action and its sub-string, is implicitly a spring, so the sub-string could match at any nested level in the source string.

4.3 The Matching Algorithm

The main loop for matching one string to another is shown in Figure 12. Cursors are also used to track the position of the search. The pattern cursor is a position just before the Code Action to be unified next. Initially the pattern cursor points to just before the very first Scope Head action. The source cursor is defined in a similar manner, except it ranges over the source string. If at some point in the search, the pattern cursor reaches the end of the pattern Code String before the source cursor comes to the end of the source Code String, then a match has been found.

Figure 12 Main matching algorithm (Step 5 of Figure 10)

1. Create an empty rollback list of Code Actions, so we can unwind the pattern string at dead ends.
2. Create an empty list of parameter bindings made, when a given code action is rolled back; we have to know which new parameter bindings (if any) to undo. This list is called the “undo binds”
3. Create a position cursor for the pattern and source strings. Both cursors are positioned at the top of the strings, just before the first Scope Head action.
4. The current spring length (used each time the cursors advance), is initially set to infinite length, so that pattern matching can start anywhere along the source string.
5. {OUTER} WHILE source cursor not at end of source string DO
6. Advance the pattern cursor to next Code Action to be specialized
7. IF pattern cursor at end of pattern string THEN
8. BREAK {out of outer loop, we have a match!}
9. ENDIF

10. Set the spring length, there may or may not have been a spring after the preceding pattern action, if so, then when advancing the source string cursor, we can advance as far ahead as needed.
11. Find the next candidate source Code Action from the source string.
12. IF no appropriate Code Action could be found in the source string THEN
13. IF the rollback list is empty THEN
14. There is nothing to backtrack to, simply abort
15. BREAK {out of outer loop}
16. ENDIF
17. Rollback to the most recently specialized pattern Code Action, undo any parameter bindings generated by unifying that Code Action, and then advance the source cursor to an alternative position to try the match again.
18. CONTINUE (restart at top of outer loop)
19. ENDIF
20. Make a temporary copy of the current pattern Code Action
21. {INNER} WHILE temporary pattern Code Action not unified DO
22. Try to unify temporary pattern code action and current source Code Action (this may introduce new parameter bindings)
23. IF unification was unsuccessful THEN
24. Accounting for the current spring length, find the next source Code Action candidate. This is the same as step 11.
25. IF no source Code Action candidate could be found THEN
26. Forget the temporary pattern Code Action and perform backtracking (jump to step 13).
27. ENDIF

28. Using the new candidate source Code Action, try to unify again.
29. CONTINUE {to top of inner loop}
30. ENDIF
31. Unification was successful. Append the temporary pattern Code Action
 to the rollback list.
32. Add any new parameter bindings to the undo binds list.
33. Advance pattern cursor to the next pattern Code Action.
34. BREAK {from inner loop}
35. ENDWHILE {INNER}
36. ENDWHILE {OUTER}

As shown in Figure 12 initialization is achieved in steps 1-4 by moving the cursors to the top of the strings, starting with an infinite spring length, and clearing the rollback. The notion of a cursor is used to represent the position in a string. The cursor always points to just before the "current" Code Action. This position may be anywhere along the outermost scope of a Code String, or any of its nested sub strings. Cursors also provide a convenient marker of progress; when the cursor on the source string comes to the end, we have a mismatch; when the cursor on the pattern string comes to the end, we have a match.

Aside from backtracking, matching is essentially moving the cursor in each string inwards if possible and then downwards, noting any parameter substitutions (bindings) needed to make the current pattern Code Action, match the current source Code Action. When a sub-string is encountered, then matching always proceeds inwards, before continuing onwards (depth first).

Backtracking moves the cursors back one position and then retries the match with the next possible source action (assuming the spring length for that position is non-zero). If that fails, then

backtracking will keep rolling back until it can find a suitable alternative path. In this way, all possible combinations are tried (a brute force approach). A basic example of cursor movement is shown in Figure 11, here the circled numbers represent the various positions where the cursors get advanced to, and Code Action unification is attempted. Positions 2 and 4 in Figure 11, show how unification leads to new parameter bindings.

The main matching loop is started in steps 5-11. The pattern cursor is advanced to the next pattern Code Action to be matched, and the source cursor is advanced to the next plausible source Code Action. If we have exhausted the pattern string, then we have match, and we can stop. If we could not find an appropriate source Code Action to try matching against, then we have to backtrack.

Backtracking is accomplished in steps 13-18. If there is nothing to rollback to, then the source string has been exhausted at the outer most scope, and we do not have a match (and can stop now). If there was a previously matched Code Action, then we can move the cursors back to that position, and try matching the next plausible source Code Action to that pattern Code Action, accounting for the spring length at that pattern position. If the string length was zero, then we can't have a match.

If a plausible source Code Action was found then unification will be attempted. In this case the pattern action and source action look like they match by type, but now the actual operators and operands must be matched. Code Action unification was discussed in Chapter 3.

Unification is attempted in step 22, inside an inner loop. The inner loop is used to ensure that if the current source Code Action does not match, then any remaining source Code Actions on the current source sub-string will be tried (always accounting for the current spring length). Also

note there they may be several plausible candidates, but only one is likely to unify properly. In the case where unification is not successful, and no further plausible candidates can be found, then backtracking is used again to restart the matching process. This is done in step 26.

Finally the last few steps of the outer loop (31-33) are used to note the pattern Code Actions as they are successfully unified. Also, any newly generated parameter bindings are noted, so that they can be undone during backtracking.

4.4 A Complete Example

To demonstrate the matching algorithm, a pattern for the standard selection sort will be used against a standard implementation of selection sort. The source Code String is shown in Figure 13, and the pattern Code String is shown in Figure 14.

Figure 13 A source Code String for the implementation of "selection sort"

Figure 14 The pattern Code String for "selection sort"

As shown in Figure 14, this pattern has 10 parameters and parameter \$5 is an implicit parameter, it is not specified directly in the pattern, but must be inferred from the complex parameters \$7, \$8 and \$9. Note also that springs have been used liberally to allow for loose matching of the pattern. When speaking of lines in the above figures, they will be referred to as the "source line" for the actual source code, and "pattern line" for the pattern lines.

The first pattern action is taken from pattern line A "for(\$1 = 0; \$1 < \$2; \$1++) ", and the first plausible source action is taken from source line 1 "for(i=0; i<n; i++)". Because the pattern action was preceded by a spring, the search for a plausible source candidate was able to search forward as needed for an iterate type action. No backtracking is necessary, so we proceed with unification of the two actions. They each have three slots, and all slots are unified easily by making simple parameter assignments. The slots to be unified are shown in Table 4. The bindings are shown in Table 5.

Table 4 Slots for unification attempt #1

\$i = 0	i = 0
\$i = \$2	i < n
\$i ++	i ++

Table 5 Bindings after unification attempt #1

\$1 is i
\$2 is n

After substituting these simple parameters back into the pattern action, unification reduces to a simple token comparison of the two Code Actions. They match, and so we can advance the cursors on both pattern and source strings.

The pattern cursor is advanced to the next action to match against on pattern line B "\$3 = \$1". The source action is taken from source line 2 "min=i". Note that since cursors are advanced depth first, we do not exhaust either string, by running off the end of the 'for' action, rather we step into its nested scope, before continuing. Although the pattern action was preceded by a spring, the spring was not needed in this case, since the next plausible source action followed the previous one directly. Before making any new bindings, the known parameters are substituted (namely "i"). Only one new simple parameter is found during unification. The unification is shown Table 6 and Table 7.

Table 6 Slots for unification attempt #2

\$3 = i	min = i
---------	---------

Table 7 Bindings after unification attempt #2

\$1 is i
\$2 is n
\$3 is min

Since the substitution of the new simple parameters reduces the unification to token comparison, and the operands all compare exactly, we can advance both the source and pattern cursors. The next pattern action is taken from pattern line C "for (\$4 = \$10; \$4 <= \$2; \$4++) ". The next source action is taken from source line 3 "for (j=i+1; j<=n; j++)". As in the first iteration a simple binding can be made for the main loop variable ('j'), however the parameter \$10 is greedy, in addition it requires the substitution of parameter \$1 in order to unify (\$10 = \$1 + 1, so \$1 must be 'i'...and it already is!). Substituting known parameters first, and then searching for simple parameters, and finally searching for greedy/complex parameters achieve all of this. The resulting unification and new bindings are shown in Table 8 and Table 9.

Table 8 Slots for unification attempt #3

\$4 = \$10	j = i + 1
\$4 <= n	j <= n
\$4 ++	j++

Table 9 Bindings after unification attempt #3

\$1 is i
\$2 is n
\$3 is min
\$4 is j
\$10 is i + 1

Using these new bindings, the two code actions unify by simple token comparison, and we can advance both pattern and source cursors. The next pattern action is taken from pattern line D "if(\$7 < \$8)". The next source action is taken from source line 4 "if(a[j] < a[min])". As before any

known parameters are substituted first, then simple parameters are searched for (nothing to do), however, greedy parameters are present in the pattern action, so when we look for implicit and greedy parameters, we find that parameter \$5 can be bound to "a", because in the context of \$7 (\$7 = \$5[\$4]), \$5 == \$a. Note that when looking for greedy/complex parameters we substitute known parameters when trying to expand a greedy parameter to its sequence of tokens/simple parameters. So, for example, \$4 and \$3 would be pasted into the expansion of both \$7 and \$8 respectively. When working on \$8, the known parameters are always replaced first, so \$8 is quickly matched using the now known parameter \$5. The scanning for implicit parameters like \$5 is done in left to right order. The resulting unification attempt and new parameter bindings are shown in Table 10 and Table 11.

Table 10 Slots for unification attempt #4

\$7 < \$8	A[j] < a[min]
-----------	---------------

Table 11 Bindings after unification attempt #4

\$1 is i
\$2 is n
\$3 is min
\$4 is j
\$5 is a
\$7 is a[j]
\$8 is a[min]
\$10 is i + 1

Using these new bindings, the code actions unify by simple token comparison. We can now advance both the source and pattern cursors. The next pattern action is taken from pattern line E "\$3 = \$4". The next source action is taken from source line 5 "min=j". In this instance all parameters of the pattern action are already bound, so only the substitution of known parameters is necessary. No new bindings are searched for or generated. Because there is no spring preceding the current pattern action, the corresponding source action can only be the one directly following the previously matched source action. In this case the action matches. If the current

source action didn't match, the lack of a spring would mean that backtracking would be needed (which with this pattern and source would still ultimately fail). Substituting for the parameters \$3 and \$4 allows unification #5 to be completed by simple token comparison.

The only task remaining is to match against the last three assignment statements. When a cursor reaches the end of a sub-string (i.e. a nested scope), the cursor is moved up to the parent string. The Code Action just after the branch point in the parent string becomes the current Code Action. Once the cursors are properly positioned, the last three statements are matched and unified using the means already discussed. The final three iterations of match/unification produce the remaining bindings for \$6 and \$9. The final parameter bindings are shown in Table 12.

Table 12 Final bindings for all parameters

\$1 is i
\$2 is n
\$3 is min
\$4 is j
\$5 is a
\$6 is t
\$7 is a[j]
\$8 is a[min]
\$9 is a[i]
\$10 is i + 1

With all parameters bound, and a complete match found (pattern was string exhausted, its cursor is at the end of the outer most string), a successful match to the "selection sort" pattern can be reported.

4.5 Backtracking Example

To demonstrate the backtracking portion of the matching process, a slightly modified version of the source code string is used. This version has an extra assignment statement after the expression action at source line 5 in Figure 13. The modified version is shown in Figure 15.

Figure 15 Modified source Code String to show backtracking...

String matching proceeds as before, except that when matching the last three assignment statements, there is a false start, because of the modified source string. In this scenario, after source line 5 is processed, source line 6 is matched against pattern line F. It should be source line 7 that gets matched against pattern line F. Source line 6 introduces a false start, and the wrong bindings are made to \$6 ("not") and \$8 ("valid").

Now when the match loop tries to match pattern line G, it finds that the cursor in the source string is at the end of a scope (just after source line 6). This is line 12 of the match algorithm in Figure 12. Since no more actions can be found in this scope of the source string, then line 17 of the main match algorithm is activated. The most recently matched action is thrown out (i.e. the false start), along with any parameter bindings made during its unification (i.e. \$6 and \$8). The source string cursor is moved to the next available action, which is source line 7. The pattern string cursor is moved back one position, which is pattern line F.

The main match algorithm jumps back to the top of its outer loop, and matching is restarted. Since we are now in a position to match the two strings as was done in the non-backtracking example, the matching completes successfully as before. If that rollback had not been sufficient, the main match algorithm would have continued rolling back Code Action matches, until a new start could be made. In the case where the matching string is completely unwound, there is no match.

4.6 Discussion

This Code String model only matches control flow, data flow is almost entirely omitted, save for the ordering of actions in a given string. This model is fuzzy in that Code Actions need not match exactly; however, when the pattern string explicitly specifies the operand names in all of

its actions, and does not use any spring connectors, then a very literal matching (token by token) occurs.

The Code String model does not account for some optimizations. For example, sometimes a programmer might interleave two loops into one to get a performance boost, this “jamming” of loops is a standard practice, yet it would be very difficult to represent this kind of optimization, since parallelism was not designed into this model. An example of jamming is shown in Figure 16.

Figure 16 Example of loop jamming

```
for(int i=0; i<3; i++) {  
    WorkA();  
}  
...  
for(int j=0; j<3; j++) {  
    WorkB();  
}  
  
↓  
  
for(int i=0, j=0; i<3, j<3; i++, j++)  
{  
    WorkA();  
    WorkB();  
}  
...
```

This model is very simple to implement and can be matched against source code very quickly without the use of complex machinery such as AST walking or regular expression machines. This model does not have all of the power of the models in TAWK or SCRUPLE, however it has enough power to model many interesting patterns, and perform very well. In addition the model is simple enough to be used in a graphical way. Further work in this area might produce a GUI where code actions can be picked from a palette and simply dropped into place in a Code String.

Very little "programming" would be involved in creating a pattern Code String to use for matching against source code.

By specifying a program plan that uses greedy parameters and very few constraining operators/operands, it would be possible to detect some instances of jammed loops. However the generalized a program plan pattern is, then the higher the chance of detecting false positives. With this in mind such generalized plan patterns should be used with care.

Chapter 5:

Code String Based Surveying

A concrete application of Code Strings is to verify the secondary question of this thesis: "Can such a model be used to survey a wide range of code and provide an empirical basis for supporting or refuting the notion of programming plans?" A survey of a broad range of real world software projects is described. The procedure and current survey software are outlined, the results of the survey are presented and discussed. In addition, a metric is introduced for counting matches of program plans (via Code String patterns) in software.

5.1 A Sample Survey

To exemplify the utility of Code Strings, the Code String matching algorithm was implemented and used as the basis for a survey tool. Given a repository of program plan patterns (discussed further below), and the source tree (the main directory on disk containing the source code) of an application to be searched, the matcher is set about searching for instances of the plans in the source tree.

After searching through the source of the application, the matcher provides a detailed report of how many instances of each plan were found. Any plan not specified in the repository, is not searched for and consequently not reported. This survey covered about 10 million lines of code, and the matching was conducted in about 2.2 hours.

The current survey software (referred to loosely as the “the matcher”) implements the matching Code String algorithm discussed in Chapter 4. The intent of the survey and the matcher is to demonstrate the capabilities of the Code String model, and show feasibility. This survey is not intended to be a complete and/or standardized taxonomy of plan patterns. This survey does not aim to provide a complete and standard distribution of matches. While goals such as these are valuable, and can be achieved with application of more plan patterns to more real world software, they are beyond the scope of this survey. It should also be noted that the matcher is not optimal, matching algorithm optimizations have been left for future research.

Performing a broad-spectrum survey is accomplished simply by invoking the matcher, with a given plan repository, on several software packages downloaded from the internet. In an attempt to survey a wide variety of source code, the following twelve (12) packages were surveyed:

- Qt-3.0.0, a popular C++ cross-platform GUI framework.
- Tcl-8.0.5, a commonly used scripting language.
- Mozilla, an open source version of the Netscape web browser.
- XEmacs-21.4.4, the open source standard programming editor.
- XMMS-1.2.5, an open source version of WinAmp (an MP3 player)
- AlepheOne-0.11.1, an open source 3D action game (a shooter).
- Bison-1.28, a parser generator that succeeded YACC.
- DDD-3.3, the open source standard GUI based debugger.
- GCC-3.0.1, the open source standard C/C++ compiler.
- Linux, everyone’s favorite kernel!
- GZip-1.2.4a, the popular open source compression tool, predecessor to BZip.

- Make-3.79, the open source standard software project dependency management/building tool.

The projects to be surveyed were intentionally chosen to be from a wide variety of application areas. Many more software projects could have been chosen, however for the purposes of demonstrating the utility of Code Strings as a survey mechanism, this small set of projects was deemed broad enough.

The matcher is an early research prototype and as yet offers no graphical interface for the input of program plans. A plan is currently specified in an ASCII file. The plan specification files are collected into a repository of plans. When searching software, all plans in the repository will be searched for. The plan pattern files have syntax similar to C/C++. The details of plan specification are expanded upon below.

The scope of this survey is limited to C/C++, however the Code Strings model could be used with other (modified) parsers/compiler that know about control flow and high level declarations of variables, procedures and such.

The current parser being used was developed at Nortel Networks for parsing large bodies of source code (>30 million lines of code) in a robust manner in the absence of header files or erroneous source code. The parser was modified to build up Code Strings as it parses a C/C++ function/method, or when it parses a plan pattern file. The pattern strings are saved to disk in a repository, and at run-time loaded into memory in their entirety. The source strings are searched as they are generated, while the parser is being run over the source tree.

5.2 Program Plan Specification (ASCII format)

In lieu of a GUI for specifying Code Strings an extended C/C++ syntax is introduced. This allows program plans to be specified in manner very similar to normal source code. Springs are specified with a hash (#) character, and simple parameters are declared like normal variables except they begin with a dollar sign (\$). Normally, C/C++ identifiers do not begin with this character. Greedy parameters are specified in the same way, except that the parameter number ends with a plus sign (+). Complex parameters are inferred from their definition when references to other simple parameters are seen. An example is shown in Figure 17, this pattern is a generalized version of the Code String in Figure 2. Parameters are used so that the operand names are not important, and a spring is used so that this pattern string can be found anywhere inside a function or method.

Figure 17 Example ASCII coding of a "swap" pattern Code String

```
void swap_1(void) {  
    $1+ -> ;  
    $2+ -> ;  
    $3+ -> ;  
    $999 -> ;  
  
    #;  
    $1 = $2;  
    $2 = $3;  
    $3 = $1;  
  
}
```

As shown in Figure 17, the parameters of the plan (\$1, \$2 and \$3) are all specified as greedy parameters. The trailing arrow is used to help recognize the declaration as plan parameters. Parameter declaration is ended by a special parameter name \$999. This ensures that any dollar sign characters appearing in the code due to macros or broken strings do not get recognized as plan parameters.

A spring is used to indicate that the plan can start anywhere within a function or method. Finally the canonical “swap” plan is specified using the parameters of the plan. Notice that there are no further springs, indicating that the swap plan must be found as a grouped chunk, and not “sprinkled” throughout a function or method.

For the purposes of this survey, each plan to search for was placed in its own specification file, however many plans may be combined into a single file if that is desired. The plan specifications used in this thesis are available on the web site <http://www.sce.carleton.ca/~garvin/codestrings>. The complete list of program plans and their specification is also provided in Appendix A.

5.3 Procedure

The plans used in this survey were:

- Accumulate, 7 different variations of an accumulator loop pattern were provided.
- Data Guard, a condition on 0 was used in this instance.
- Doubly Linked List Deletion, two variations of deleting an item from a doubly linked list were provided.
- Doubly Linked List Insertion, two variations of inserting an item into a doubly linked list were provided.
- List Traversal, a common pointer based traversal of a linked list pattern was provided.
- Selection Sort, a pattern for the standard selection sort algorithm was provided.
- Sentinel Loop, four different variations of loop termination patterns were provided.
- Swap, the canonical swap pattern was provided.

With the source trees for all twelve sample projects in hand, as well as a repository of program plans, that matcher was then invoked on each sample project, using the pattern repository. The results of the survey are detailed below. The matcher has built in features for searching a source tree for C/C++ files. When a C/C++ file is found, it will automatically be searched for program plans. This built in source file detection feature allows a source tree to be specified simply by providing the top level directory containing the source tree.

The survey was conducted on a Linux (RedHat 6.1) based laptop computer, a Dell Inspiron with an Intel Pentium III running at 400Mhz. All disk access occurred via the internal hard drive. This computer is considered to be "off the shelf". All timing information was gathered via the `timex` command, and the computer had no other programs running when the analysis was conducted.

5.4 Results

The summary results are shown in Table 13. All timing results are shown wall clock seconds.

Table 13 Summary of survey.

Project	Total Analysis Time	Total Parse Time	Total Lines Parsed	Total Files Parsed	Total Patterns Tried	Avg. File Parse Time	Avg. File Analysis Time
Qt	461.32	173.26	816,032	1,311	19	0.086	0.33
Tcl	64.93	22.88	128,437	128	19	0.15	0.50
Mozilla	2011.12	744.81	3,238,259	6,288	19	0.08	0.28
XEmacs	211.95	78.42	376,359	334	19	0.11	0.60
XMMS	49.28	16.28	63,356	147	19	0.07	0.31
AlephOne	62.99	25.27	108,547	145	19	0.10	0.41
Bison	10.02	3.29	16,270	37	19	0.06	0.25
DDD	108.4	42.38	228,436	443	19	0.06	0.23
GCC	737.65	284.4	1,514,059	5,246	19	0.04	0.13
Gzip	5.04	1.79	9,110	21	19	0.05	0.22
Linux	2088.86	809.11	3,407,713	4,539	19	0.10	0.43
Make	18.73	6.48	33,837	34	19	0.11	.53

These performance results are depicted in a more intuitive way in Figure 18 below. Note that the trend lines were generated by the standard least squares method.

Figure 18 Analysis and parsing performance across all sample projects

The summary of plan instances found is shown in Table 14, Table 15, Table 16, and Table 17. These tables contain the raw number of matches found; no processing has been done to normalize the numbers. Data normalization is introduced after Table 17

Table 14 Pattern instances found (patterns 1-5)

Project	list_traversal_1 (LT1)	data_guard_1 (DG1)	Swap (SWP)	accumulate14 (ACC14)	sentinel_loop_2 (SL2)
Qt	4	8	12	30	122
Tcl	8	2	0	13	60
Mozilla	38	65	54	155	532
XEmacs	5	7	6	29	77
XMMS	0	0	6	4	16
AlephOne	3	1	2	4	37
Bison	2	1	3	3	17
DDD	4	3	6	25	112
GCC	33	9	38	48	259

Gzip	3	0	0	1	13
Linux	105	80	61	261	819
Make	3	1	1	7	44

Table 15 Pattern instances found (patterns 6-10)

Project	sentinel_loop_1 (SL1)	sentinel_loop_3 (SL3)	sentinel_loop_4 (SL4)	accumulate_1 (ACC1)	accumulate_2 (ACC2)
Qt	5	20	33	308	265
Tcl	1	16	6	73	31
Mozilla	10	184	99	1920	708
Xemacs	2	36	7	241	79
XMMS	1	4	1	65	46
AlephOne	0	2	8	69	68
Bison	1	2	3	50	15
DDD	3	8	6	110	61
GCC	12	90	78	549	357
Gzip	0	3	4	37	8
Linux	42	165	89	2893	1161
Make	5	10	4	68	19

Table 16 Pattern instances found (patterns 11-15)

Project	accumulate 3 (ACC3)	accumulate 7 (ACC7)	accumulate 6 (ACC6)	dlist item insert 1 (DII1)	accumulate 15 (ACC15)
Qt	189	28	211	1	26
Tcl	59	19	12	0	14
Mozilla	1257	245	652	1	191
Xemacs	201	36	75	0	41
XMMS	39	6	20	0	20
AlephOne	69	6	12	0	8
Bison	21	4	15	0	5
DDD	83	9	69	4	12
GCC	440	82	197	19	50
Gzip	28	4	16	0	3
Linux	2492	177	692	5	445
Make	42	15	9	1	9

Table 17 Pattern instances found (patterns 16-19)

Project	Dlist_item_delete (DID)	Dlist_item_insert (DII)	Dlist_item_delete_1 (DID1)	selection_sort (SS)
Qt	0	0	0	0
Tcl	0	0	0	0
Mozilla	9	0	0	0
Xemacs	0	0	0	0
XMMS	0	0	0	0
AlephOne	0	0	0	0
Bison	0	0	0	0
DDD	2	0	0	0

GCC	8	1	1	0
Czip	0	0	0	0
Linux	5	0	0	0
Make	0	0	0	0

Table 18, and Table 19 shows the normalized results from Table 14, Table 15, Table 16, and Table 17. Normalization is achieved by taking the number of instances found, dividing by the number of lines of code in a given project and then multiplying by 1000 to obtain a plan metric. This metric approximates the percentage of the source code where the given plan is at least started. Lines of code are used as the normalization weighting, rather than number of files, since files can be of variable length.

Table 18 Normalized summary of plan instances found (plans 1-10).

Project	LT1	DG1	SWP	ACC14	SL2	SL1	SL3	SL4	ACC1	ACC2
Qt	0.005	0.01	0.015	0.037	0.15	0.006	0.025	0.04	0.377	0.325
Tcl	0.062	0.016	0	0.101	0.467	0.008	0.125	0.047	0.568	0.241
Mozilla	0.012	0.02	0.017	0.048	0.164	0.003	0.057	0.031	0.593	0.219
Xemacs	0.013	0.019	0.016	0.077	0.205	0.005	0.096	0.019	0.64	0.21
XMMS	0	0	0.095	0.063	0.253	0.016	0.063	0.016	1.026	0.726
AlephOne	0.028	0.009	0.018	0.037	0.341	0	0.018	0.074	0.636	0.626
Bison	0.123	0.061	0.184	0.184	1.045	0.061	0.123	0.184	3.073	0.922
DDD	0.018	0.013	0.026	0.109	0.49	0.013	0.035	0.026	0.482	0.267
GCC	0.022	0.006	0.025	0.032	0.171	0.008	0.059	0.052	0.363	0.236
Czip	0.329	0	0	0.11	1.427	0	0.329	0.439	4.061	0.878
Linux	0.031	0.023	0.018	0.077	0.24	0.012	0.048	0.026	0.849	0.341
Make	0.089	0.03	0.03	0.207	1.3	0.148	0.296	0.118	2.01	0.562

Table 19 Normalized summary of plan instances found (plans 11-19).

Project	ACC3	ACC7	ACC6	DH1	ACC15	DID	DH	DID1	SS
Qt	0.232	0.034	0.259	0.001	0.032	0	0	0	0
Tcl	0.459	0.148	0.093	0	0.109	0	0	0	0
Mozilla	0.388	0.076	0.201	0	0.059	0.003	0	0	0
Xemacs	0.534	0.096	0.199	0	0.109	0	0	0	0
XMMS	0.616	0.095	0.316	0	0.316	0	0	0	0
AlephOne	0.636	0.055	0.111	0	0.074	0	0	0	0
Bison	1.291	0.246	0.922	0	0.307	0	0	0	0
DDD	0.363	0.039	0.302	0.018	0.053	0.009	0	0	0
GCC	0.291	0.054	0.13	0.013	0.033	0.005	0.001	0.001	0
Gzip	3.074	0.439	1.756	0	0.329	0	0	0	0
Linux	0.731	0.052	0.203	0.001	0.131	0.001	0	0	0
Make	1.241	0.443	0.266	0.03	0.266	0	0	0	0

In order to better compare the numbers of plans found in each project, the match numbers are normalized (as discussed above), and the average number of matches for each plans calculated. The resulting average number of plans found is summarized in Table 20. A Student's T distribution was used to calculate the 95% confidence intervals.

Table 20 Average of normalized plan hits across all projects

Plan/Pattern	Average Hits (Normalized)	95% Conf. (-)	95% Conf. (+)
LT1	0.061	0.013	0.109
DG1	0.017	0.009	0.026
SWP	0.037	0.01	0.064
SL1	0.023	0.001	0.045
SL2	0.521	0.28	0.763
SL3	0.106	0.053	0.159
SL4	0.089	0.027	0.152
ACC1	1.223	0.601	1.846
ACC2	0.463	0.324	0.602
ACC3	0.821	0.413	1.23
ACC6	0.397	0.147	0.646
ACC7	0.148	0.071	0.225
ACC14	0.09	0.061	0.12
ACC15	0.151	0.09	0.213
DID	0.002	0	0.003
DID1	0	0	0
DII	0	0	0
DII1	0.005	0	0.01
SS	0	0	0

The results from Table 20 are shown in a more institutive manner in Figure 19.

Figure 19 Average number of plans found across all projects.

5.5 Discussion

The Code String matching algorithm is a brute force approach. As shown in Figure 18, this approach appears to scale well. A greater repository of patterns and larger sampling of software would be beneficial though, in ensuring there is no hidden cases of degraded backtracking that could hinder performance.

Normalization of the plan match counts is achieved through a crude weighting by lines of code. The resulting match counts exhibit a high degree of variability. Clearly a better match count metric could be designed. A better metric might be based on the set of variations for a plan, and not each variation. Even with this shortcoming though, the resulting match counts (shown in Figure 19), show a significant jump in counts for sentinel and accumulation based loops. This empirically derived result seems to reaffirm the intuitive notion that loops are the most common

control structure used by programmers. It is not clear yet that loops are more common than assignment type statements. A broader repository of plans was not attempted in this thesis, since the aim was only to demonstrate the utility of the Code Strings model.

While list traversal, item deletion and removal seem to be much less prevalent, the plan repository used contains fewer variations of these plans than it does for sentinel and accumulation based loops. This lack of definition in the plan repository could be favoring the loop plans. Future work should include a much more thorough set of plans, to achieve more balanced results.

The single example of a complex algorithm (SS - Selection Sort), found no matches. This is likely due to the lack of any variations, most programs are likely to include common optimizations that are not specified in the given "text book" selection sort plan.

In summary the results of the survey are interesting in that they provide a strong empirical basis for the intuition that loops play a central role in programming. However, a better plan match metric needs to be designed that can account for plan variations, and help reduce variability. Further a more exhaustive plan repository would be crucial in developing metrics that could be considered strongly unbiased.

Chapter 6:

Conclusions and Further Work

A visual model of program plan specification has been developed. A matching algorithm was developed, allowing this model to be used to search large amounts of software for program plan instances. A new plan instance counting metric was developed. In addition the matching software and the initial program plan repository has been made available. This chapter summarizes the findings, and suggests some future directions for developing Code Strings further.

6.1 Contributions and Findings

A new visual model of program plans was presented. A matching algorithm was presented that allows the Code Strings model to be used for searching source code for instances of program plans. By way of a broad survey of existing real world code, and a sample set of programming plans, the matching algorithm appeared to scale, and provide a means of searching for programming plans. Since no other broad software survey is known to have been conducted before, the survey data is a new contribution to the ESP field.

The survey conducted shows that programming plans have a distribution; some are more commonly used than others (in particular loop based plans), however a more thorough set of plan patterns would be useful in eliminating any bias due to the distribution of plan patterns searched for.

The survey work also generated a “plan-hits” metric. This metric is used to normalize the counts of plans found, and make the numbers more suitable for comparison among software projects.

This approach involves much less programming by an end user than existing methods. Since this approach is based on control flow structure, it is very sensitive to structural changes in the source code. The result of this is that often many variations of a programming plan have to be given as patterns to search for, because any one of the variations might appear in the source code. In addition source code fragments that combine plans (also known as “jamming”, shown in Figure 16) are unlikely to be properly identified, again due to the control flow based nature of the model.

Matching software was designed and implemented. The matching software was then successfully applied to a broad set of real world programming projects. Such a system could be used to easily test a given body of source code for the existence of a hypothetical programming plan.

The drawbacks of the current survey software are that it lacks heuristics for optimization in searching (future work), and also lacks a graphical interface for input of Code String models. Verification of the precision of this model was not conducted on a known sample of test cases from well known source code, say from a standard teaching text.

6.2 Further Work

Among the most important directions for this work:

- A web site for standardized plan patterns, and search results from various plan matching techniques. This should provide a taxonomy of program plans and techniques, as well as more standardized results.

- A GUI for specifying programming plan patterns, and further development of the graphical language used for Code Strings.
- The “hits” metric could be improved and redesigned to be more proportional, and have a better correlation between the size/complexity of the plan being searched for, and the size/complexity of the software project being searched. The hits metric could also have nesting level incorporated into it, or the length of the function where the hit happened.
- The survey could be improved by expanding the patterns repository to include a broader selection of plans, and more variations of each. Essentially a better taxonomy of plans.
- Code Strings are useful for finding control flow patterns, but are not very useful for finding data flow patterns. Given the presence of design idioms like “jamming”, a model that incorporated some data dependencies could be very useful. A model that can span function calls would also be useful. For example sometimes the contents of a loop are coded as the body of a single function call.
- Code Strings are currently only being applied to the problem of finding plan patterns, however, Code Strings could be used to look for signatures other than control flow. For example, signatures of machine instructions in binary code, or say the flow of control through tasks in a multitasked system. Each action would be a lightweight task.
- To address the verification limitation, the source code of the survey should be randomized line by line within functions/methods. Then the survey should be run again. The results should show a program plan distribution very different from the distribution already found. If not, then the current model may be detecting a side effect such as complexity, rather than the plans themselves.
- The matching power of Code Strings code be extended by allowing the slots in Code String patterns to support some simple forms of regular expressions such as alternative patterns.

BIBLIOGRAPHY

- [Ball95] Ball L., Ormerod T., "*Structured Opportunistic Processing in Design: A Critical Discussion*", 1995, International Journal of Human Computer Studies Vol. 43, pp 131-151, 1995
- [Bellamy90] Bellamy R., Gilmore D., "*Programming Plans: Internal or External Structures*", 1990, Lines of Thinking Vol. 2, 1990
- [Burnstein97] Burnstein I., Roberson K., Saner F., Mirza A., Tubaishat A., "*A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool*", 1997,
- [Davies90] Davies S., "*The Nature and Development of Programming Plans*", 1990, International Journal of Man Machine Studies, Vol. 32, pp 461-481, 1990
- [Gellenbeck91] Gellenbeck E., Cook C., "*An Investigation of Procedure and Variable Names as Beacons during Program Comprehension*", 1991, Empirical Studies of Programming, 4th conf. pp 65-81, 1991
- [Gilmore88] Gilmore D., Green T., "*Programming Plans and Programming Expertise*", 1988, Quarterly Journal of Experimental Psychology, 40A (3) pp 423-442, 1988
- [Griswold96] Griswold W., Atkinson D. and McCurry C., "*Fast, Flexible Syntactic Pattern Matching on Processing*", 1996, Proceedings of the IEEE Fourth Workshop on Program Comprehension, March 1996.
- [Harandi88] Harandi M., Ning J., "*PAT: A Knowledge-Based Program Analysis Tool*", 1988
- [Johnson85] Johnson W., Soloway E., "*PROUST: Knowledge-Based Program Understanding*", 1985, IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, pp 267-275, March, 1985
- [Kozaczynski92] Kozaczynski W., Ning J., Sarver T., "*Program Concept Recognition*", 1992
- [Murphy96] Murphy G., Notkin D., "*Lightweight Lexical Source Model Extraction*", 1996, ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3., pp 262-292, July 1996
- [Paltheu97] Paltheu S., Greer J., McCalla G., "*Cliché Recognition in Legacy Software: A Scalable Knowledge-Based Approach*", 1997
- [Parker87] Parker, J. and Hendley, B. "*The Universe Program Development Environment*", 1987, Proceedings of INTERACT'87.
- [Paul94] Paul, S. and Prakas, A. "*A Framework for Source Code Search Using Program Patterns*", 1994, IEEE Transactions on Software Engineering, Vol. 20, No. 6, June 1994.
- [Pennington87] Pennington N., "*Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*", 1987, Cognitive Psychology Vol 19 pp 295-341, 1987.

- [Quilici93] Quilici A., "*A Hybrid Approach to Recognizing Programming Plans*", 1993
- [Quilici94] Quilici A., "*A Memory Based Approach to Recognizing Programming Plans*", Communications of the ACM, 1994, Vol. 37, No. 5, May 1994
- [Quilici97] Quilici A., Woods S., Yongjun Z., "*New Experiments With A Constraint-Based Approach to Program Plan Matching*", 1997
- [Rich88] Rich, C. and Waters, R. "*The Programmer's Apprentice Project: A Research Overview*", 1988, IEEE Computer, pp 11-25, Nov 1988.
- [Rich90] Rich C., Willis L., "*Recognizing a Program's Design: A Graph-Parsing Approach*", 1990, IEEE Software, Jan 1990.
- [Soloway84] Soloway, E. and Ehrlich, K. "*Empirical Studies of Programming Knowledge*", 1984, IEEE Transactions on Software Engineering, SE-10, pp 595-609
- [Soloway84] Soloway E., Ehrlich K., "*Empirical Studies of Programming Knowledge*", 1985, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September, pp 595-609, 1984
- [Spohrer85] Spohrer J., Soloway E., Pope E., "*A Goal/Plan Analysis of Buggy Pascal Programs*", 1985, Human Computer Interaction Vol. 1, pp 163-207, 1985
- [Woods95] Woods S., Qiang Y., "*Program Understanding as Constraint Satisfaction*", 1995
- [Woods96] Woods S., Quilici A., "*Some Experiments Towards Understanding How Program Plan Recognition Algorithms Scale*", 1996

Appendix A: Survey Program Plans

This appendix lists all program plans used in the software survey. They are formatted in ASCII form, as described in Chapter 5. Each program plan was specified as a single function, and placed in a single file. All of the plans are concatenated here into a list of functions.

6.3 ASCII Plan Specifications From Survey

All of the plans used in the survey are detailed here. There is one subsection for each plan used. The plan shorthand names used in the survey charts are also used as the subsection titles below.

When the plans were specified by hand, some of the function names were reused when creating variations of plan patterns. This does not invalidate the search results in any way since the function names are simply placeholders. However this lazy naming practice can make reading some of the plan specifications confusing. When in doubt please check the contents of the plan pattern.

In the plan specifications the token \$999 is used as a special sentinel marker to tell the parser to stop looking for plan pattern parameter declarations. In some plan patterns the usage of a plan parameter can look exactly like the declaration format. The \$999 marker forces the detection of parameter declarations to stop so that in those cases the plan parameter usage will be treated as a normal expression. As with the naming the usage of the sentinel marker was lazy and applied in some cases where it wasn't actually necessary. This does not invalidate any of the survey results, but can make the plan patterns more confusing to read.

Most program plan patterns used greedy parameters. This is because it was not expected that real source code would use single token operands. It was expected that in most cases, the

operands of a given operator would be a multi token expression. The result is that the program plan patterns constrain searching mostly by control flow and operator placement within the pattern, since the operands can be largely arbitrary.

6.4 ACC1

```
void accumulate1(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
  
    #;  
    $1 = $2;  
    #;  
    while() {  
        #;  
        $1++;  
    }  
}
```

6.5 ACC14

```
void accumulate4(void) {  
  
    $1+ -> ;  
  
    #;  
  
    while($1 > 0) {  
        #;  
        $1--;  
    }  
}
```

6.6 ACC15

```
void accumulate4(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
    $3 -> ;  
  
    #;  
  
    while($1 > $3) {  
        #;  
        $1 -= $2;  
    }  
}
```

6.7 ACC2

```
void accumulate2(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
    $3+ -> ;  
  
    $999 -> ;  
  
    #;  
  
    $1 = $2;  
  
    #;  
  
    for() {  
  
        #;  
  
        $1 += $3;  
    }  
}
```

6.8 ACC3

```
void accumulate3(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
    $3+ -> ;  
  
    #;  
    $1 = $2;  
    #;  
    while() {  
        #;  
        $1 += $3;  
    }  
}
```

6.9 ACC6

```
void accumulate6(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
  
    #;  
    while($1 < $2) {  
        #;  
        $1++;  
    }  
}
```

6.10 ACC7

```
void accumulate6(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
  
    #;  
    while(*$1 == $2) {  
        #;  
        $1++;  
    }  
}
```

6.11 DG1

```
void data_guard1(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
    $999 -> ;  
  
    #;  
  
    if ($1 > 0) {  
        $2 = $1;  
    }  
}
```

6.12 DID1

```
void dlist_item_delete1(void) {  
  
    $1 -> ; // the object pointer  
  
    $2 -> ; // the previous pointer  
  
    $3 -> ; // the next pointer  
  
    $999 -> ; // end of parameter decls  
  
    $1->$2->$3 = $1->$3;  
    $1->$3->$2 = $1->$2;  
  
}
```

6.13 DID

```
void delete_item1(void) {  
  
    $1 -> ;  
    $2 -> ;  
    $3 -> ;  
    $4 -> ;
```

```

$5 -> ;

$999 -> ;

$1 = $2->$3;
$4 = $2->$5;

#;

$1->$5 = $2->$5;
$4->$3 = $2->$3;

}

```

6.14 DII

```

void dlist_item_insert1(void) {

    $1 -> ;
    $2 -> ;
    $3 -> ;
    $4 -> ;
    $5 -> ;
    $999 -> ;

    #;

    $1 = $2->$3;
    $4 = $2->$5;

    #;

    $1->$5 = $2;
    $4->$3 = $2;

}

```

6.15 DIH

```

void dlist_item_insert1(void) {

    $1 -> ;
    $2 -> ;
    $3 -> ;
    $999 -> ;

    #;

    $1->$2->$3 = $1;
    $1->$3->$2 = $1;

}

```

6.16 LT1

```
void list_traverse1(void) {  
  
    $1+ -> ;  
    $2+ -> ;  
    $3+ -> ;  
    $4+ -> ;  
  
    #;  
  
    for ($1 = $2; $1 != NULL; $1 = $3) {  
  
        #;  
  
        $3 = $1->$4;  
    }  
}
```

6.17 SS

```
void selection_sort_1(void) {  
  
    $1 -> ;  
    $2 -> ;  
    $3 -> ;  
    $4 -> ;  
    $5 -> ;  
    $6 -> ;  
    $7 -> $5[$4];  
    $8 -> $5[$3];  
    $9 -> $5[$1];  
    $10 -> $1+1;  
  
    #;  
  
    for($1 = 0; $1 < $2; $1++) {  
  
        #;  
  
        $3 = $1;  
  
        #;  
  
        for($4 = $10; $4 <= $2; $4++) {  
  
            #;  
  
            if($7 < $8)  
                $3 = $4;  
            #;  
        }  
  
        #;  
    }  
}
```

```

    $6 = $8; $8 = $9; $9 = $6;
}
}

```

6.18 SL1

```

void sentinel_loop1(void) {

    $1+ -> ;
    $2+ -> ;
    $3 -> ;
    $4 -> ;
    $999 -> ;

    #;

    $1 = $2;

    #;

    while ($1->$3 != $4) {

        #;

        $1 = $1->$3;
    }
}

```

6.19 SL2

```

void sentinel_loop2(void) {

    $1+ -> ;
    $2+ -> ;
    $3+ -> ;
    $999 -> ;

    #;

    while ($1 != $2) {

        #;

        $1 = $3;
    }
}

```

6.20 SL3

```

void sentinel_loop3(void) {

    $1+ -> ;
    $2+ -> ;
    $3+ -> ;

```

```

$999 -> ;

#;

while () {

    #;

    $1 = $2;

    #;

    if($1 == $3) {
        break;
    }
}

```

6.21 SL4

```

void sentinel_loop3(void) {

    $1+ -> ;
    $2+ -> ;
    $3+ -> ;
    $4+ -> ;
    $999 -> ;

    #;

    for($1 = $2; $1 != $3; $1++) {

        #;

        if($1 == $4) {
            break;
        }
    }
}

```

6.22 SWP

```

void swap_1(void) {

    $1+ -> ;
    $2+ -> ;
    $3+ -> ;
    $999 -> ;

    #;
    $1 = $2;
    $2 = $3;
    $3 = $1;
}

```