A Framework for Source Code Search Using Program Patterns

Santanu Paul, *Member, IEEE*, and Atul Prakash, *Member, IEEE*

Abstract— **For maintainers involved in understanding and reengineering large software, locating source code fragments that match certain patterns is a critical task. Existing solutions to the problem are few, and they either involve manual, painstaking scans of the source code using tools based on regular expressions, or the use of large, integrated software engineering environments that include simple pattern-based query processors in their toolkits. We present a framework in which pattern languages are used to specify interesting code features. The pattern languages are derived by extending the source programming language with pattern-matching symbols. We describe SCRUPLE, a finite state machine-based source code search tool, that efficiently implements this framework. We also present experimental performance results obtained from a SCRUPLE prototype, and the user interface of a source code browser built on top of SCRUPLE.**

Index Terms—**Reverse engineering, software maintenance, software reengineering, program understanding, pattern matching, query language.**

I. Introduction

THERE has been a growing interest in the software engineering community to develop techniques that help software engineers search through large amounts of source code to locate relevant information [2], [3], [7], [10], [23], [28]. Facilities to search through source code can be useful in several situations:

1) *Reengineering code:* To reengineer software, a software engineer may need to detect the existence of a repetitive code, so that it can be replaced with calls to a single procedure. The engineer may also wish to detect code with potentially poor structure, such as procedures with too many levels of nested loops, or statements that use a sequence of *if-then-else*'s where *case* statements might be more appropriate. Tools that employ automatic *program transformation* techniques for reengineering have identified source code search as their primary performance bottleneck as well [19].
2) *Making queries on programs:* A software developer may need to locate all lines in which a procedure is called (for instance, when the procedure interface is to be changed), a variable is modified (when it is found to have incorrect value), etc. When a bug is found in the algorithm used in one section of code, a developer may need to locate other sections of code that use a similar algorithm since those sections could have the same bug.
3) *Understanding programs:* To understand programs, programmers often make a hypothesis (enlightened or otherwise) a bout what the program does, scrutinize the source code to confirm the hypothesis, and revise the hypothesis based on what is discovered during the scrutiny [6]. For example, a programmer may hypothesize that a program needs to do matrix multiplication, and then look for a code fragment with three nested iterative statements:

```
iterative (...)
  { ......
     iterative (...)
     { ......
          iterative (...)
          {
                  .........
          }
     }
  }
iterative: while, for, do-while (for C)
```

A tool that can help find such code fragments can be very helpful in validating or revising the programmer's hypothesis.

In the examples mentioned above, there are two types of entities: 1) a *specification* of the features being looked for, and 2) *matches*, which are the entities or fragments in the source code that fit the specification. Of course, specifications of the features are expressed informally in the above examples. If we could provide an easy-to-use formalism to express them, then there is a hope of providing tools to help with finding the matches.

The formalism used to express the specifications must allow users flexibility regarding the degree of precision in the specification. For instance, a programmer trying to locate a matrix multiplication routine may wish to specify only a control structure containing three nested loops, omitting details of contents of the loops, whereas a developer trying to locate all the exact copies of a certain piece of code may wish to use the code piece itself as the specification.

In this paper, we describe a framework for alleviating search problems similar to those outlined above. In the proposed framework, specifications are written using a *pattern language*,

Manuscript received April 15, 1993; revised February, 1994. Recommended by D. Parnas. This work was supported by a Research Fellowship from IBM Canada, Ltd.

The authors are with the Software Systems Research Laboratory, Dept. of Electrical Engineering & Computer Science, University of Michigan, Ann Arbor, MI 48105 USA; e-mail: santanu@eecs.umich.edu, aprakash@eecs.umich.edu.

IEEE Log Number 9401706.

which is an extension of the programming language being used. The extensions include a set of symbols that can be used as substitutes for syntactic entities in the programming language. When a specification is written using one or more of these symbols, it plays the role of an abstract template which can potentially match different code fragments. If no symbol is used, the specification consists only of constructs which are valid in the programming language, which effectively makes it a valid code fragment in itself, and hence leads to only precise matches.

We have prototyped a system, SCRUPLE, for two languages C and PL/AS (a PL/1 variant) to experiment with our framework. SCRUPLE's pattern matching engine automatically finds source code entities or fragments that match the specifications written in SCRUPLE's pattern language. The engine proceeds by transforming the source code to a syntax-tree representation and transforming the user's specification to a special-purpose nondeterministic automaton. It then finds the matches by running the generated automaton on the source code's syntax tree representation.

Section II compares our work with other techniques that are used for doing search through source code. Section III describes the key features of our framework. Section IV discusses the SCRUPLE pattern language used to search through C programs. Section V outlines the architecture and algorithms of the SCRUPLE runtime system that locates code fragments that match specified patterns. Section VI discusses performance figures based on a suite of sample queries. Finally, Sections VII and VIII present our conclusions and plans for future work.

II. Comparison With Other Tools

A. Tools Based on Regular Expressions

Why don't we just use the tools of the `grep` family in UNIX to specify patterns and extract the matches from the source code? Both `grep` and `egrep` can match regular expressions, which is a powerful mechanism for pattern matching [16]. They both, however, have the following limitations that make them unsuitable for our purposes.

1) Writing certain code specifications using `grep` can be difficult, and sometimes impossible. For example, if one wished to look for a sequence of two statements—a `while` followed by a `for` (both with arbitrary conditions and bodies) *at the same level of nesting*—then a `grep` specification might be
```
    .*while.*for
```
2) Unfortunately, this specification would also match code where the `for` is in the body of the `while`, or where the `while` or `for` occur within comments, which is clearly not what is desired.
3) A serious problem with `grep`-like tools involves writing specifications for data declarations. Consider writing a specification for a declaration of an integer variable x and a character variable$c$. Since the ordering of declarations is usually unimportant, they could appear in any order in the source code. A regular expression for such a specification would have to be of the following form:
```
(.*int[ ]*x[ ]*;.*char[ ]*c[ ]*;)|
(.*char[ ]*c[ ]*;.*int[ ]*x[ ]*;)
```
The complexity of specifying patterns containing more than two declarations is obvious.
4) The existing implementations of `grep` family tools do not support the matching of `newline` characters. This is a serious drawback in the context of source code search where most meaningful patterns of code span multiple lines.

Other tools that employ regular expressions for pattern matching include `ed`, `sed`, and `awk`. `ed` and `sed` [16] are text editors which allow find-replace facilities on regular expressions (which in the case of `sed` can span multiple lines), essentially treating the source code as a character stream. `Awk` [1] is a pattern matching and processing language that views its input as a stream of records. While the record abstraction is an improvement over character streams, it does not significantly help in source code search because records are inadequate for modeling the complexities in source code.

The shortcomings of these tools in the context of source code search stem from two facts: 1) the inappropriateness of character streams or record streams as models of source code and 2) the inadequacy of regular expressions as a source code query language. For source code search, the source code should be modeled using its abstract syntax representation. Similarly, the query language must permit nested and recursive patterns. As a result, the use of these tools makes queries unnatural, clumsy, and often impossible.

B. Browsers and Program Databases

Omega [20], CIA [7], CIA++ [10], Microscope [3], Rigi [23], and SCAN [2] are examples of tools that are designed especially for making queries on code and help with code browsing. They typically generate a *program database* consisting of entities such as file names, function names, and variable names, with well-defined relations between them. Queries can then be made on the program database to retrieve information such as names of functions that call a specified function, global variables used by a specified function, etc.

Though the above tools are powerful and useful, they support only a limited range of queries—those that are based on a entity-relation-attribute view of programs. In contrast, our scheme takes a structural approach to source code search, while supporting many of the features of the abovementioned systems.

C. Tools to Detect Plagiarism

Tools that detect plagiarism in programs (the problem is relevant in the case of student assignments) [4], [11], [21] are usually based on software metrics such as Halstead's [12]. Another category of tools [14] utilize the *static execution tree* (the call graph) of a program to determine the 'fingerprint' of a program, and use the latter to decide whether it has been copied from another program. The primary limitation in all such systems is that the comparison depends on statistical information, which makes them unsuitable for writing structural specifications. We are proposing techniques which give a greater degree of control over the type of matching desired, and allow matching over a part of the program (such as a procedure, declarations, or statements) rather than a matching between two programs.

D. Program Transformation Tools

Systems have been developed to manipulate source code through program transformation [19]. The purpose is to automate software tasks such as development, modification, and correction. Transformations are specified using *rules*. The left-hand side of a rule is a code pattern. The right-hand side consists of actions that must be performed if the left hand side matches.

The TXL system [8] converts code written in a dialect language into code in the base language. This is accomplished by transforming the parse tree derived using the dialect grammar into a parse tree of the base language grammar, and extracting a new program from the latter. Apparently, a recursive tree matching algorithm is employed by the tree transformer. The specifics of the algorithm and its complexity are unavailable. The ASCENT system [9] generates program transformers to convert programs written in an application-specific language into those written in a general-purpose language, by transforming parse trees.

In contrast to the implicit tree matching in the above systems, the REFINE system [28] offers an explicit pattern-based query language to manipulate source code [18]. Program reengineering systems [19] have used REFINE to write program transformation rules. The lefthand side of these rules are usually source code patterns written using the REFINE pattern language. The REFINE source code model used is based on the abstract syntax of the programming language. The pattern language supports a rich set of features including named and unnamed wildcards, matching of high level data types such as sets and sequences, etc. The pattern matching algorithm approach is based on tree matching [31]. SCRUPLE has similarities with REFINE in its interactive pattern language based approach to source code search, but uses a finite state machine based pattern recognizer to efficiently find matches.

III. Features of Our Approach

Our solution to the search problem addresses many of the limitations of existing schemes effectively. Some of the important features of our approach are the following:

1) The pattern language used is an extended version of the underlying programming language. In particular, most code fragments are valid patterns in the pattern language. This makes learning to write patterns quite simple.
2) The pattern matching approach is syntax-directed instead of character-based. Making the matching approach syntax-directed provides a higher abstraction to the user in terms of specifying patterns.
3) A wide range of patterns, which are either very difficult or impossible to express using just regular expressions, can be expressed quite easily in SCRUPLE.
4) Pattern language gives the user substantial control over expressing the precision of specifications.
5) The search for a match can span multiple lines. In fact, the search is independent of the formatting of the source code.
6) An efficient recognizer of syntactic/structural code patterns expressed using the pattern language has been designed, and a prototype system built. The recognizer is based on well-known principles in automata theory.

IV. The Pattern Language

In SCRUPLE, users use a pattern language to specify high-level patterns for making queries on the source code. So far, we have designed pattern languages for C and PL/AS. The SCRUPLE system has been tested on real PL/AS programs. The C version of the system is in a prototype stage. In view of the popularity of C as a programming language, we are currently implementing a complete SCRUPLE system for it.

The pattern language in our framework is an extension of the source code programming language. The extensions include a set of symbols that can be used as substitutes for syntactic entities in the programming language. When a specification is written using one or more of these symbols, it plays the role of an abstract template which can potentially match different code fragments.

The pattern symbols that lend the pattern language its expressive power can be classified into four broad categories: 1) wildcards for syntactic entities, 2) wildcards for collections of syntactic entities, 3) named wildcards, and 4) additional features provided to allow complex queries regarding nesting, references to identifiers, constraints and restrictions on the names and entities to which wildcards get bound, etc.

To illustrate our approach, we give an overview of the pattern symbols in a sample pattern language for C. More detailed descriptions of the pattern language are available in [24], [26].

A. Wildcards for Syntactic Entities

Queries about source code (written in imperative languages) often pertain to the programming language constructs such as statements, variable declarations, type declarations, expressions, functions, etc. To make such queries possible, we introduce pattern symbols for such constructs. Table I lists these pattern symbols. These pattern symbols can substitute as wildcards in the patterns used to express queries about the source code.

A few queries that can be expressed using these symbols are described below.

- *Query: Find all* `while` *statements where the condition of the* `while` *statement is a relational expression of the form not-equal-to zero.*

  Pattern:

  ```
  while (#!= 0) @;
  ```

  The goal might be to replace `(#!= 0)` by simply `(#)`.

TABLE I
WILDCARDS FOR SYNTACTIC ENTITIES

	Syntactic Entity	Pattern Symbol
1	declaration	$d
2	type	$t
3	variable	$v
4	function	$f
5	expression	#
6	statement	@

TABLE II
WILDCARDS FOR COLLECTIONS OF SYNTACTIC ENTITIES

	Entity Collection	Pattern Symbol	Semantics
1	declarations	$*d	SET
2	variables	$*v	SET
3	expressions	#*	SET
4	statements	@*	SEQUENCE

- *Query: Find all occurrences of three consecutive* `if` *statements.*

  Pattern:
  ```
  if # @;
  if # @;
  if # @;
  ```
  The goal might be to locate potential candidates for `switch` statements.
- *Query: Find all* `if` *statements where '=' has been mistakenly used in place of '==' in the condition.*

  Pattern:
  ```
  if (# = #) @;
  ```
  The goal is to locate a common source of bugs in C programs—namely the use of the assignment operator instead of the equality operator in a relational expression.
- *Query: Find all declarations of the variable x.*

  Pattern:
  ```
  $t x;
  ```
  The goal might be to improve readability of the code by introducing mnemonic names instead of x.

We chose the current symbols based on our perceptions of what maintainers typically look for. If queries requiring pattern matching on other syntactic entities were required, such syntactic entities could be added easily to the pattern language without changing the basic design of the system.

B. Wildcards for Collections of Syntactic Entities

In addition to the basic pattern symbols introduced in the previous section, symbols that represent collections of these entities are also necessary. For example, the user might be interested in matching a collection of declarations or statements of arbitrary size. The pattern symbols for collections of syntactic entities are listed in Table II.

Collections of syntactic entities can have semantics which are relevant to the problem of pattern matching. These semantics must hold for the pattern symbols that represent these collections. For example, we know that in the source code of a C program, the order of declarations in a group of declarations usually does not matter. Therefore, the two declaration groups shown below are treated as identical for the purposes of pattern matching.

```
Group I        Group II
int x;         char s;
char s;        int x;
```

However, in the case of statements, the order is important and for the purposes of pattern matching the following two groups are not identical.

```
Group I                   Group II
if (x > 0) y = y*x;       y = y+1;
y = y+1;                  if (x>0) y = y*x;
```

Based on these examples, it is apparent that syntactic entity collections are of specific types. A group of declarations form a set, a group of statements form a sequence, a group of expressions form a set, and a group of variables form a set. Consequently, $*d is a wildcard for a set of arbitrary declarations, #* for a set of arbitrary expressions, @* for a sequence of arbitrary statements, and $*v for a set of arbitrary variables. For the purposes of pattern matching, the matching rules applied to a collection of syntactic entities are determined by its type.

A few queries that can be expressed using these symbols are described below:

- *Query: Find all statements which are procedure calls.*

  Pattern:
  ```
  $f(#*);
  ```
- *Query: Find all functions which return values of type* `COMPLEXNUMBER`.

  Pattern:
  ```
  COMPLEXNUMBER $ f_1($*v) $*d {@*;}
  ```
  The goal is to locate all functions which return data of a certain type.
- *Query: Find a sequence of statements such that three or more* `if` *statements occur, possibly with other statements between them.*

  Pattern:
  ```
  if # @;
  @*;
  if # @;
  @*;
  if # @;
  ```
  The goal could be to locate code sections with high cyclomatic complexity [22]: in this case there are eight possible paths through the source code.
- *Query: Find a set of declarations, one of which is a declaration of a variable* `maxval` *of* `type int`.

  Pattern:
  ```
  int maxval;
  $*d;
  ```

TABLE III
NAMED WILDCARDS

	Entity	Pattern Symbol
1	declaration	$d_{name}
2	declaration set	$*d_{name}
3	type	$t_{name}
4	variable	$v_{name}
5	variable set	$*v_{name}
6	function	$f_{name}
7	expression	#_{name}
8	expression set	#*_{name}
9	statement	@_{name}
10	statement seq.	@*_{name}

TABLE IV
OTHER PATTERN SYMBOLS

	Pattern Symbol	Meaning
1	@{{...{}...}}	A statement with depth of nesting equal to the number of {.
2	@{**}	A statement with arbitrary nesting depth.
3	@[stmt-type1 | stmt-type2...]	Any of the specified statements
4	@<id-1,id-2,· · ·>,#<· · ·>,$f<· · ·>	refers to/uses identifiers

C. Named Wildcards

The pattern symbols described in the last two sections make the pattern language reasonably expressive; however, there is still a large class of queries which cannot be expressed using just these features. Consider the query: *Find all instances where a variable of type* `integer` *is incremented by 1*. It is clear that the query needs to be expressed as a combination of two simple patterns—the first pattern expressing a variable of type `integer`, and the second pattern expressing that the *same* variable is incremented by 1. To make such queries possible, the concept of named wildcards are introduced. Named wildcards imply bindings, and can be used to express constraints within patterns, and to restrict the matching of one part of a pattern based on that of another part. The list of named wildcards are given in Table III.

The query mentioned earlier:
Find all instances where a variable of type `integer` *is incremented by 1* can now be expressed as:

Pattern:
```
int $v_1;
$v_1 = $v_1 + 1;
```

Using the concept of semantically equivalent statements described in Section IV-F, this query can be used to match statements of the type $v_1 + +; as well.

A few other queries that can be expressed using named wildcards are described below:

- *Query: Find situations where the values of two variables are being swapped.*

  Pattern:
  ```
  $v_tmp = $v_x;
  @*;
  $v_x= $v_y;
  @*;
  $v_y = $v_tmp;
  ```
  The goal is to recognize a swapping *plan* or *cliche*.

- *Query: Find all* `struct` *declarations containing a field whose type is recursively defined.*

  Pattern:
  ```
  $ t_1 {
    $*d;
    $ t_1 *$v;
  }
  ```
  The goal may be to detect the various linked lists used in the program.

D. Additional Power

This section covers the remaining features of our pattern language. The inclusion of these features makes the pattern language considerably more expressive. Table IV lists the pattern symbols for these features.

The symbols 1 and 2 in Table IV permit the specification of nesting information in patterns. Symbol 3 imposes constraints on potential matches. Symbol 4 is intended to specify usage information (i.e., whether or not a certain identifier is used or referred to within a statement, expression, or function).

A few queries that can be expressed using these symbols are described below:

- *Query: Find all functions that have references to the identifier* `xmax`.

  Pattern:
  ```
  $t $f_x ⟨xmax⟩ ($v*) { @* };
  ```
  The goal may be to examine the accessibility of the identifier `xmax` from various functions.

- *Query: Find a structure of three nested loops.*

  Pattern:
  ```
  @[while | for | dowhile ]{@*;
    @[while | for | dowhile ]{@*;
      @[while | for | dowhile ]{@*;
      }
    }
  }
  ```
  The goal is to look for solutions to matrix multiplication, path closure, etc.

E. Writing a Pattern

Using the symbols mentioned in the previous sections, patterns can be written. The text of a pattern has two sections. In the first section, any restrictions that apply to variable, type or function names are declared. In the second section, the actual pattern of the code being looked for is described. The two sections are separated by %%.

Pattern

```
$f_1 = '*max*'
%%
$t_1 $f_1($*v)
$*d
{*
  @[while|dowhile|for] {*
                 if ($v_2[#] > $v_3)
                      $v_3 = $v_2[#];
               *}
*}
```

A match

```
int find_max(int_arr, N)
int int_arr[ ];
int N;
{
   int i;
   int maxstore;
   maxstore = int_arr[0];
   for (i=1;i<N;i++) {
   if (int_arr[i] > maxstore)
        maxstore = int_arr[i];
   }
   return(maxstore);
}
```

Fig. 1. Pattern for finding the maximum in an array of integers.

Pattern

```
while ($v_1<25) {
      $v_3[$v_1] = $f_5($v_1);
      $v_1++;
}
```

Match

```
do {
   squares[x] = squareof(x);
   x++;
} while (x < 25);
```

Fig. 2. Matching after transformation to canonical forms.

Suppose a maintainer wants to locate a function that finds the maximum value in an array of integers. He suspects that the name of the function contains the substring "max" in it. In addition, the maintainer assumes that the function has a store for the current maximum, which is updated as necessary as the entire array of integers is scanned from left to right. After the scan of the array is done, the store contains the maximum value in the array.

Using this knowledge, the maintainer may come up with the pattern in Fig. 1, for which the code on the right is a match. $v_3, $f_1, and $v_2 are bound to `maxstore`, `find_max`, and `int_arr` respectively.

F. Matching Equivalent Statements

The abstraction provided by the features of the pattern language described so far can be further enhanced by introducing low-level semantics to the matching mechanism. Consider a pattern consisting of an iterative statement (say `while`). It may happen that the source code contains a `do-while` statement that has a body and a terminating condition matching that of the `while` statement in the pattern. In such situations, we may want a match to be reported. An example is given in Fig. 2.

Low-level semantics of this type can be introduced into the matching mechanism by mapping equivalent constructs into *canonical forms*. Such a representation scheme would map `while`, `for` , and `do-while` statements in C to a *common* iterative form. The patterns which use these constructs would also be transformed to the corresponding common iterative form. The problem of matching the `while` and the `do-while` statements would thus reduce to comparing their canonical forms. At this point, we have not implemented this facility in our prototype.

V. SCRUPLE System Architecture

The SCRUPLE run-time system searches the source code for matches. The program source code is transformed by a *source parser* into a data structure called the *attributed syntax tree* (AST), which is based on the attributed dependency graph model described in [2]. The pattern (or query) specified by the user is transformed by a *pattern parser* into an automaton called *code pattern automaton* (CPA). CPAs are special-purpose nondeterministic finite state automata. The formal definition of a CPA is given in Section V-B. The major components of the system are shown in Fig. 3.

After the AST and CPA have been generated, a CPA *interpreter* runs the CPA with the AST as input. A match occurs whenever the CPA reaches a final state. The interpreter maintains information about bindings of named wildcards in data structures called *binding tables*.

In our prototype, the source and pattern parsers are written manually using a high-level programming language (C). It is however possible to generate the parsers automatically from a high-level description of the abstract syntax of the source code programming language [25], [28].

We now describe the SCRUPLE architecture in some more detail.

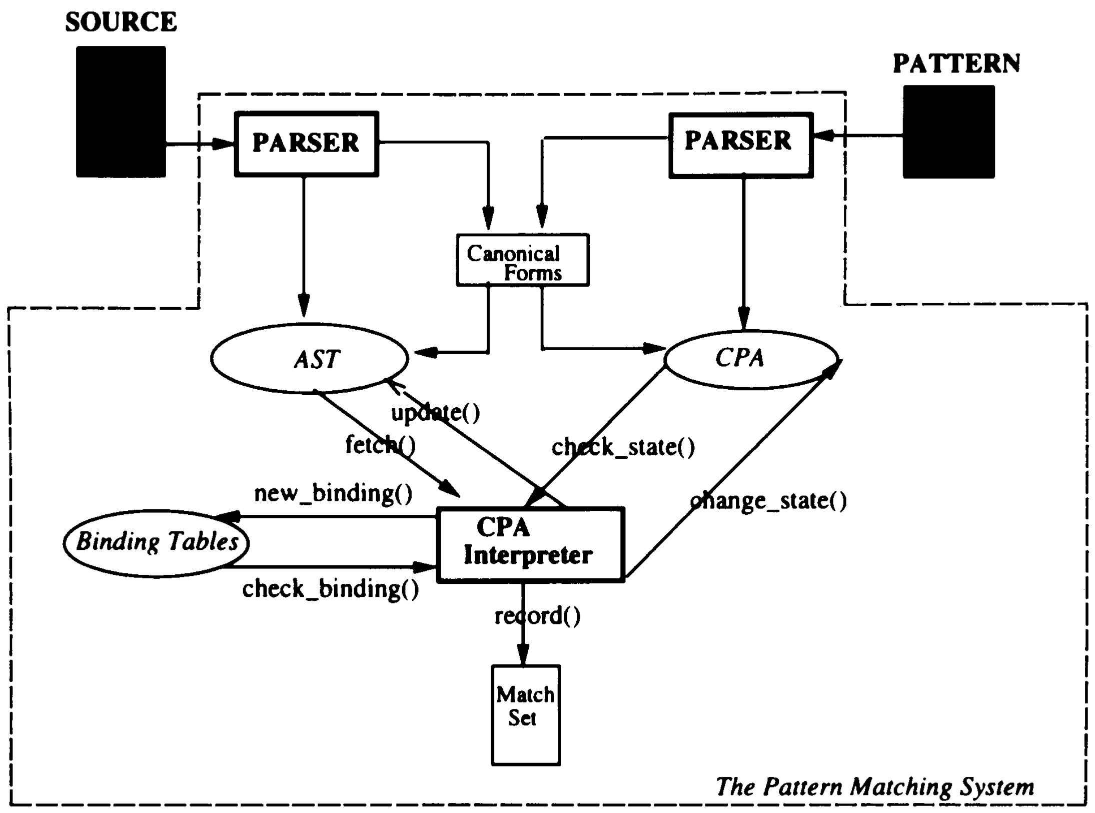


Fig. 3. The architecture of the SCRUPLE system.

A. AST: Attributed Syntax Trees

The source parser compiles the source code (on which the search is to be carried out) into an AST. The AST is a tree data structure that captures the abstract syntax of source code (see Fig. 4). The nodes of the AST represent the entities of the program, along with attributes that contain information about the entities. The entities can be *functions, declarations, statements, expressions,* or other terminals and nonterminals of the source code language grammar. The use of abstract syntax representations in software engineering has been documented in [27].

B. CPA: Code Pattern Automata

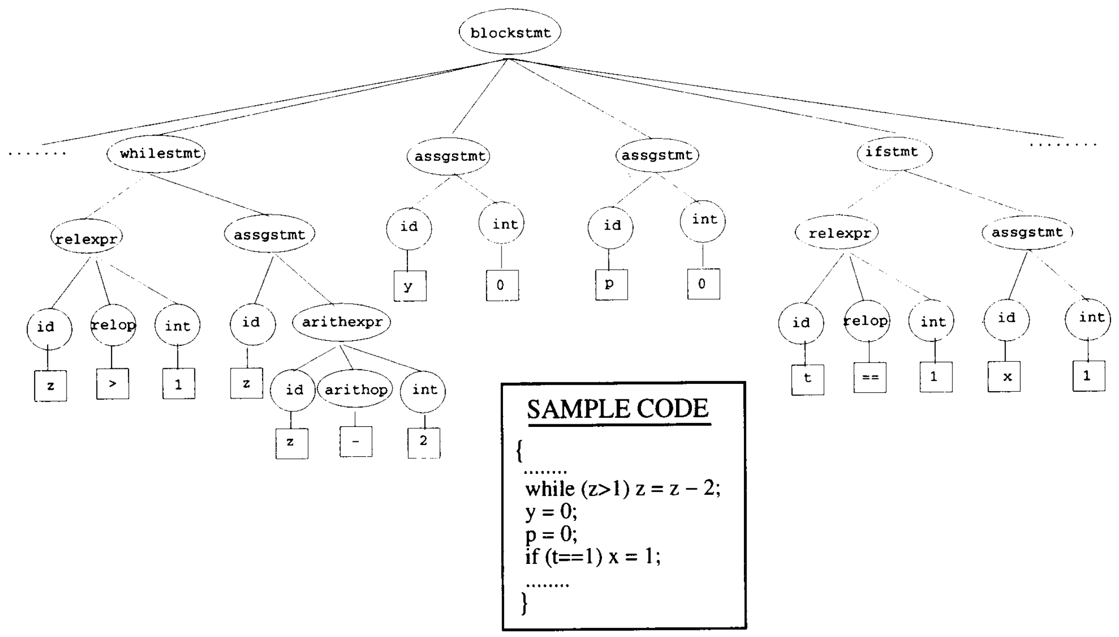
The pattern parser compiles a user-specified pattern into a CPA, an extended nondeterministic finite state automaton. Finite state automata are a good basis for designing efficient pattern detection algorithms. Nondeterminism permits the detection of wildcards such as @*, #*, etc., and also makes it possible to explore multiple *potential matches* simultaneously. The input to the CPA is an AST. Ordinary finite state machines cannot be run with a syntax tree as input, and must be extended.

The definition of a CPA evolves from that of a classical nondeterministic finite state machine [13]. Two key extensions to the classical model are introduced. First, the input alphabet consists of syntactic elements of the source code, i.e., nodes of the AST (terminals and nonterminals). Secondly, transition arcs between CPA states contain explicit information about the "next" AST node that must be seen by the new state. Essentially, this is a mechanism of navigating through the AST to generate the correct input stream for each match being explored by the CPA.

Formally, a CPA is a 6-tuple of the form $\langle Q, \Sigma, A, \Gamma, q_0, F \rangle$ where we have the following:

1) Q is the set of states.
2) Σ is the input alphabet consisting of nodes of the AST that represent syntactic elements. We define $\Sigma = \Sigma_N \cup \Sigma_L$, where Σ_N represents the internal nodes of the AST and Σ_L represents the leaves.
3) A is the set of AST navigation functions given by $A = \{$*moveto_leftchild, moveto_rightsibling, moveto_parent*$\}$. The semantics of the functions are apparent from their names. These are are used by the CPA interpreter to navigate through the AST to generate input streams for the CPA.
4) Γ is the set of transition functions given by $\Gamma = \{$*CAT, VAL*$\}$, where we have the following:

   - $CAT : Q \times \Sigma_N \longrightarrow 2^{Q \times A^+}$
     Arc Label:
     $CAT\ \langle category \rangle : ACTION : \langle actions \rangle$



SAMPLE CODE

```
{
........
while (z>1) z = z – 2;
y = 0;
p = 0;
if (t==1) x = 1;
........
}
```

Fig. 4. Sample attributed syntax tree used in SCRUPLE.

The *CAT* arc specifies the syntactic category to which the input must belong. Consider the inheritance hierarchy of the source code language. If T is a type, then for all S that are subtypes of T, an arc labeled *CAT*$\langle T \rangle$ can be traversed if an instance of type S occurs in the input stream. For example, if a `whilestmt` is seen, an arc labeled *CAT*$\langle$*stmt*$\rangle$ can be traversed.

$2^{Q \times A^+}$ expresses nondeterminism in the automata transitions. There could be several valid transitions from the given state for the same input. For each possible transition, a new CPA state is reached and one or more actions to advance the current node pointer in the AST input are taken. The choice of the transition taken is nondeterministic.

- $VAL : Q \times \Sigma_L \longrightarrow 2^{Q \times A^+}$
  Arc Label:
  $VAL\ \langle value \rangle ; ACTION : \langle actions \rangle$

  A *VAL* arc can be traversed when the value seen in the input stream matches the $\langle$*value*$\rangle$ on the arc.

5) q_0 in Q is the initial state.
6) $F \subseteq Q$ is the set of accepting states.

An AST is accepted as input by a CPA if there exists a sequence of transitions, corresponding to the input AST, that leads from the initial state to some accepting state.

Fig. 5 shows the CPA corresponding to the following pattern:

```
while (#) @;
@*;
if (#) x = 1;
```

Note how the arc corresponding to @* (self-referencing arc on node 3 in Fig. 5) introduces nondeterminism in the CPA.

C. The CPA Interpreter

The CPA interpreter simulates a CPA on an AST and produces a match set. The interpreter simulates the necessary state transitions of the CPA, and also moves the current AST node pointer to the next input node, as specified by actions on the transition arcs. The only type of actions used on transitions in our automata are the following: *moveto_leftchild, moveto_rightsibling*, and *moveto_parent*$^+$; *moveto_rightsibling*. Restricting the actions to just the above types implies that no node in the input AST will be seen twice, thus ensuring termination of the interpreter algorithm. A match is found if and when the CPA reaches a final state. A simulation of the CPA in Fig. 5 on the AST fragment in Fig. 4 is shown in Fig. 6.

Let us consider the interpreter algorithm under the simplifying assumption that the pattern has no named wildcards. Since the CPA is a nondeterministic machine, the interpreter uses a queue-like data structure to simulate its nondeterminism using a deterministic algorithm. The queue consists of elements called **states**, where a state is a 2-tuple of the form $\langle$*CPA_node, AST_node*$\rangle$. The queue is partitioned into segments of *input equivalent* states (states that will consume the same AST node as input or, equivalently, have the same value for *AST_node*). The interpreter reads the next state to be simulated off the front of the queue and, in case of

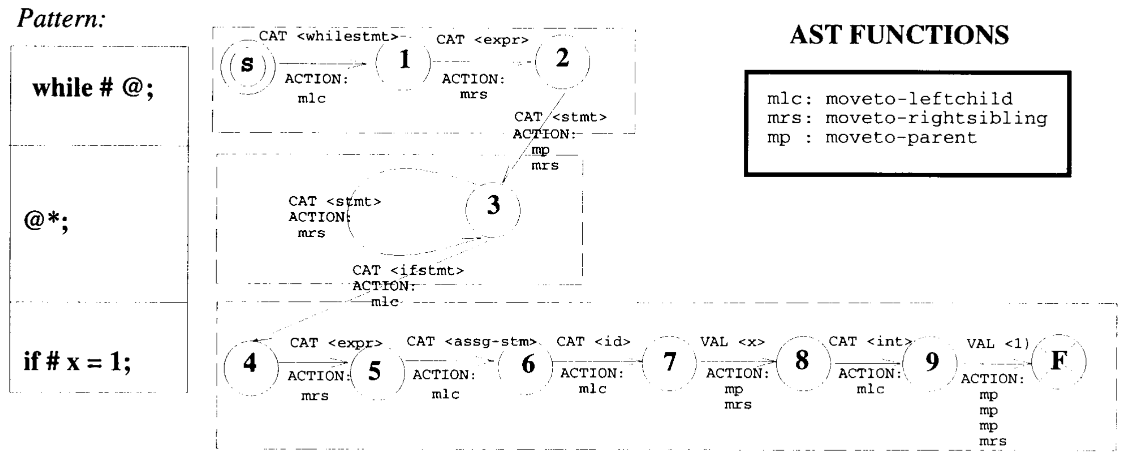


Fig. 5. Code pattern automaton.

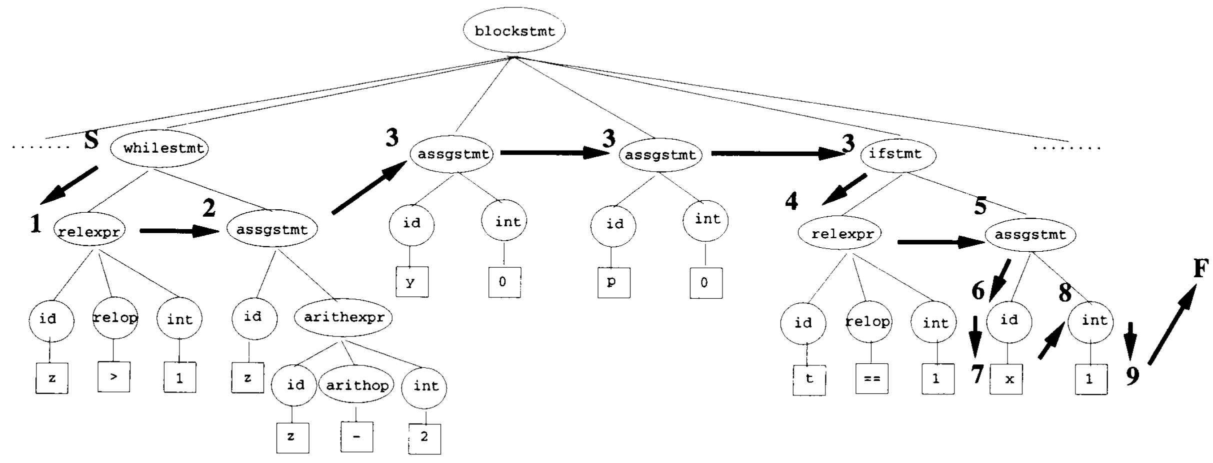


Fig. 6. CPA simulation (the numbers on the nodes show the current CPA state).

possible transitions, inserts the next states into their appropriate segments, creating new segments if necessary. A match occurs when a state with *CPA_node* $\in F$ arrives at the front of the queue.

To control the complexity of the simulation, duplicate states (states with identical values of *CPA_node* and *AST_node*) are prevented from being inserted into the queue. In addition, the segments in the queue are ordered by increasing preorder numbers of the corresponding AST nodes. If the number of AST nodes and CPA nodes is given by N and M, respectively, then the two conditions mentioned above ensure that no more than NM states are examined during the course of a CPA simulation. For each state, the time taken to process a next state is bounded by $(N + M)$, where N is the time to locate its appropriate segment, and M is the time to examine the states in that segment for duplicates. Each state may have at most M next states, hence the total complexity of the simulation is proportional to $NM^2(N + M)$. Typically, $N \gg M$, hence the algorithm is $O(N^2)$. This is of the same worst-case complexity, in the length of the input, as algorithms used to match regular expressions in text strings [29].

When the pattern has named wildcards, the interpreter uses *binding tables* to keep track of the bindings between named wildcards in the pattern and actual names seen in the source code. Because the matching is nondeterministic, there may be more than one match being explored at any given time. A unique binding table is associated with each such exploration, and hence more than one binding table may be active during the simulation of the CPA. The binding table for a given exploration stores only the bindings *relevant* to the exploration.

We point out here that traditional and efficient pattern matching algorithms such as the Knuth, Morris and Pratt algorithm [17], the Boyer and Moore algorithm [5], and the Rabin and Karp algorithm [15] were not used to solve our problem because they do not match regular expressions.

D. Performance Issues

Managing the bindings of named wildcards causes problems in terms of algorithm complexity. If the user is interested in all possible binding scenarios for a pattern, in the worst-case, pattern matching using named wildcards can be expensive. For example, consider the pattern:

```
int $v_1;
int $v_2;
```

If there exist n variables in the code of type `int`, then we have $n(n-1)$ possible bindings for the pattern.

To manage the combinatorial blow-up caused by named wildcards, we are currently looking into heuristics that can speed up the search for matches. In the absence of any named wildcards in the pattern, the pattern matching engine adopts a *shortest match* strategy. This means that if a pattern can potentially match two multiple code fragments starting at the same position, only the shortest one is matched. If named wildcards exist, a *shortest match for each binding* strategy is adopted. We implement this strategy by associating a binding table with each current state. The presence of too many named wildcards can still be combinatorially explosive; however, our experience indicates that such patterns are rare in program understanding and browsing tasks. We are looking for additional heuristics to control the complexity of named wildcards matching as a part of our ongoing research.

Another strategy adopted currently in SCRUPLE to deal with named wildcards is to match the statements and expressions in the pattern before attempting to match the data declarations. This is a useful approach because, statement or expression matching (a sequence matching problem) being less expensive than declaration matching (a set matching problem), the bindings generated in the first phase can be used to perform a controlled search for matching declarations.

The performance of the SCRUPLE system could also be improved by using additional node attributes to record program fingerprints [14], metric informations [4], [12], etc.

VI. Experimental Results

Fig. 7 shows the interface of the SCRUPLE prototype for PL/AS. The interface provides commands for selecting a file on which matching is to be done, passing the file to SCRUPLE for parsing, creating and editing patterns, and searching for matches. The pattern corresponding to a query used by the user is shown in the *pattern window* (the lower right box). The source code file being queried is shown in the *source window* (lower left box). When the user chooses the option "Match Pattern" from the pull-down menu associated with the action "Match" (on the action bar), the SCRUPLE runtime system executes, and a summary of the matches found are returned in the *results window* (upper left box). If the user now clicks on one of these matches, say Match 2, the matched source code is automatically highlighted in the source window. The binding of the pattern symbol(s) for the selected match is shown in the *bindings window* (upper right box). The user can also navigate around the source code inspecting other matches and bindings using the buttons **Next Highlight**, **Prev Highlight**, etc.

Using our PL/AS prototype, we have shown that the pattern language of SCRUPLE is powerful enough to make queries similar to those possible using other browsing systems such as CIA [7] and SCAN [2], in addition to expressing a wide range of structural queries that are impossible using other systems.

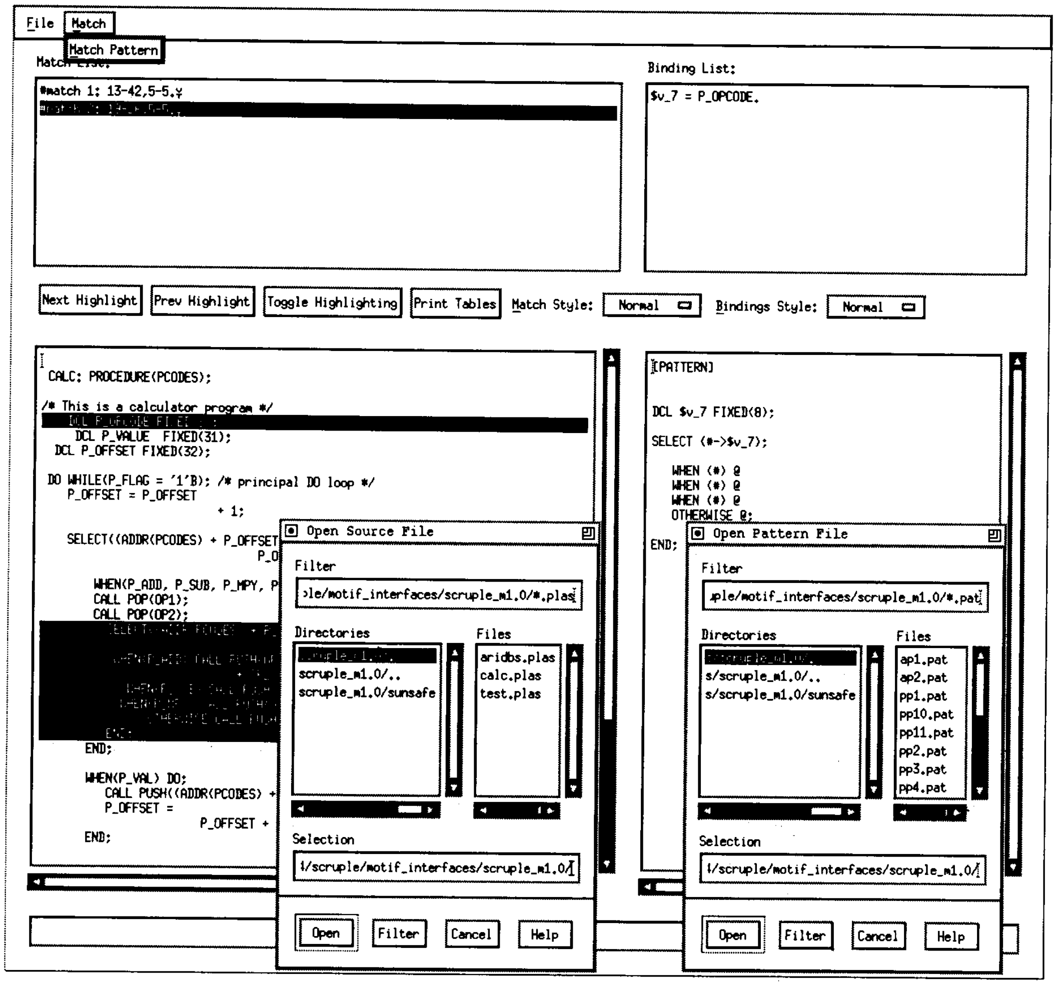
The tests described in this section were conducted on a 9,600 line PL/AS program using a Sun SPARC station. A suite of five queries was used. The queries were:

- Q1: Find all the places in the source where any procedure call is made:
  ```
  CALL $p_1(#*);
  ```
- Q2: Find all the places where the procedure ARIDMGE is called:
  ```
  CALL ARIDMGE_1(#*);
  ```
- Q3: Find all code fragments with the following nested loop structure:
  ```
  DO;
    @*;
    IF # THEN DO;
         @*;
    END;
    @*;
  END;
  ```
- Q4: Find all statements in the source code where a variable is being incremented:
  ```
  $v_1 = $v_1 + #;
  ```
- Q5: Find a sequence of three consecutive IF statements:
  ```
  IF # THEN @;
  IF # THEN @;
  IF # THEN @;
  ```

Q1 and Q2 are examples of simple queries supported by systems like `grep`, CIA, SCAN, etc. [2], [7], [10]. Q3, Q4, and Q5 are examples of queries that SCRUPLE is adept at handling, but are not handled easily by other tools.

Additionally, we also tested the performance of `grep` on Q1, Q2, and a query very similar to Q4 (which involved searching for a pattern of the form: `[A-Z]*[ ]*=[]*[A-Z]*[ ]*+.*;`). Q3 and Q5 are unsuitable for `grep` because the instances of code that will match these patterns will very likely span multiple lines, and `grep` would fail to detect them. The performance figures for SCRUPLE and `grep` on the queries are shown in Table V. The numbers exclude the time taken by SCRUPLE to parse the source code into an AST, since it is incurred only once at the beginning of a query session.

Our experience with SCRUPLE shows that it is also an effective tool for expressing queries that are typical in a software maintenance or reengineering situation. Authors of program transformation systems for code reengineering [19] have independently arrived at similar conclusions about REFINE-like pattern languages. We interpret this as a reaffirmation of our



```
#match 1: 13-42,5-5.
[illegible]
```

```
$v_7 = P_OPCODE.
```

```
 CALC: PROCEDURE(PCODES);

/* This is a calculator program */
   [illegible]
    DCL P_VALUE  FIXED(31);
  DCL P_OFFSET FIXED(32);

 DO WHILE(P_FLAG = '1'B); /* principal DO loop */
    P_OFFSET = P_OFFSET
                    + 1;

    SELECT((ADDR(PCODES) + P_OFFSET
                          P_O

       WHEN(P_ADD, P_SUB, P_MPY, P
       CALL POP(OP1);
       CALL POP(OP2);
       [illegible]
     END;

     WHEN(P_VAL) DO;
       CALL PUSH((ADDR(PCODES) +
       P_OFFSET =
                   P_OFFSET +
     END;
```

```
>le/motif_interfaces/scruple_m1.0/*.plas
```

```
[illegible]
scruple_m1.0/..
scruple_m1.0/sunsafe
```

```
aridbs.plas
calc.plas
test.plas
```

```
i/scruple/motif_interfaces/scruple_m1.0/
```

```
[PATTERN]


DCL $v_7 FIXED(8);

SELECT (#->$v_7);

   WHEN (#) @
   WHEN (#) @
   WHEN (#) @
   OTHERWISE @;

END;
```

```
sple/motif_interfaces/scruple_m1.0/*.pat
```

```
[illegible]
s/scruple_m1.0/..
s/scruple_m1.0/sunsafe
```

```
ap1.pat
ap2.pat
pp1.pat
pp10.pat
pp11.pat
pp2.pat
pp3.pat
pp4.pat
```

```
i/scruple/motif_interfaces/scruple_m1.0/
```

Fig. 7. User-interface of a browsing application that uses PL/AS version of SCRUPLE.

work in SCRUPLE, where the goal is to build a powerful pattern language.

VII. CONCLUSION

We have described a framework for specifying high-level patterns in terms of programming language constructs. High-level patterns provide a natural way to express structural features which are either difficult or impossible using `grep` like languages. A pattern language is an extension of the underlying programming language, which makes it relatively easy to use. The pattern matching is *syntax-driven* as opposed to *character-based*, which provides a better abstraction to the user, and results in an effective search method. To validate our framework, SCRUPLE prototypes have been built for C and PL/AS and demonstrated at conferences. Preliminary results using the prototypes show that SCRUPLE simplifies the task of locating complex code fragments. The strength of SCRUPLE lies in the combination of a good source representation, a powerful pattern matching engine, and a high-level query language.

The work described in this paper is based on the premise that structural patterns are a useful and interesting means of investigating source code. This hypothesis needs to be rigorously validated. Two aspects to this validation exist. First, detailed studies need to be carried out on the kinds of structural queries that arise in software maintenance. Finally, SCRUPLE must be tested widely on large-scale software systems.

VIII. FUTURE WORK

To make SCRUPLE more powerful, research needs to be done in the area of canonical representations so that simple semantic equivalence can be established for language constructs. The problem of semantic equivalence is difficult, and starting with simple programming constructs may be a useful approach. Automatic generation of SCRUPLE implementations for different programming languages is also under investigation [25].

TABLE V
PERFORMANCE (IN SECONDS)

		Real Time	Matches
Q1	SCRUPLE	1.06	335
	grep	1.45	335
Q2	SCRUPLE	1.12	270
	grep	1.47	270
Q3	SCRUPLE	3.70	250
	grep	X	X
Q4	SCRUPLE	2.64	140
Q4'	grep	12.96	145
Q5	SCRUPLE	3.65	160
	grep	X	X

Extensions to the current SCRUPLE system are being considered. A library of frequently used patterns, including those available on other program browsing systems [2], [7], [10], will enhance the utility of SCRUPLE. Alternative ways of letting the user navigate through the match set are being considered. We also wish to introduce a general mechanism for query composition using which more complex queries can be constructed out of simpler ones. Related to this is the idea of *query pipelining*, where the output of one query can be the input to another. This can be useful in query refinement and will improve the efficiency of the search.

New application areas for pattern-based query processing are emerging. Two areas that we have identified are distributed debugging and multimedia databases. In distributed debugging, the domain of search is the *event history* of executing processes, and queries can be expressed as patterns of communication behavior between these processes. In multimedia databases, data is often parsed according to a simple grammar. In the case of a video database of CNN newsclips [30], clips are organized into higher level syntactic entities like stories, episodes, and so on. For the purposes of querying, the video database can be likened to a syntax tree of clips, stories, and episodes, and pattern matching techniques similar to SCRUPLE can be applied.

ACKNOWLEDGMENT

We acknowledge the efforts of B. Monroe who transformed SCRUPLE from a command line tool to a usable system by building an X-Windows interface for it. We thank M. Knister and R. Al-Zoubi for their valuable comments on different aspects of this work. E. Buss, J. Henshaw, and J. Slonim at the Centre for Advanced Studies, IBM Canada Ltd., have been a source of great help. Finally, we thank the editor and referees for their insightful suggestions.

REFERENCES

[1] A. V. Aho, B. W. Kernighan and P. J. Weinberger. *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988.

[2] R. Al-Zoubi and A. Prakash, "Software change analysis via attributed dependency graphs," Tech. Rep CSE-TR-95-91, Dept. of EECS, Univ. of Michigan, May 1991. Also in *Software Maintenance*, to be published.

[3] J. Ambras and V. O'Day, "Microscope: A program analysis system," in *Proc. 20th Hawaii Inter. Conf. Syst. Sci.*,1987, pp. 460–468.

[4] H. L. Berghel and D. L. Sallach, "Measurements of program similarity in identical tasking environments," *SIGPLAN Notices*, vol. 19, no. 8, 1984.

[5] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol., 20, no. 10, 1977, pp. 762–772.

[6] R. Brooks, "Towards a theory of comprehension of computer programs," *Int. J. Man-Machine Studies*, vol. 18, pp. 543–55, 1983.

[7] Y. Chen, M. Y. Nishimoto and C. V. Ramamoorthy, "The C information abstraction system," *IEEE Trans. Software Eng.*, vol. 16, no. 3, pp. 325–334, Mar. 1990.

[8] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," *Comput. Lang.*, vol. 16, no. 1, pp. 97–107, 1991.

[9] D. Garlan, L. Cai and R. L. Nord, "A transformational approach to generating application-specific environments," in *Proc. SIGSOFT*, 1992.

[10] J. E. Grass, "Object-Oriented Design Archaeology with CIA++," *Comput. Syst. J. USENIX Assn.*, vol. 5, no. 1 pp. 5–67, Winter 1992.

[11] S. Grier, "A Tool that detects plagiarism in Pascal Programs," *SIGSCE Bulletin*, vol. 13, no. 1, 1981.

[12] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[13] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[14] H. T. Jankowitz, "Detecting plagiarism in student pascal programs," *The Comput. J.*, vol. 31, no. 1, 1988.

[15] R. M. Karp and M. O. Rabin, "Efficient randomized pattern matching algorithm," Tech. Rep. TR-31-81, Aiken Computation Lab., Harvard Univ., 1981.

[16] B. W. Kernighan and R. Pike, *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[17] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. , 6, no. 2, pp. 323–350, 1977.

[18] G. B. Kotik and L. Z. Markosian, "Automating software analysis and testing using a program transformation system," in *Proc. ACM SIGSOFT*, 1989, pp. 75–84.

[19] W. Kozaczynsky, J. Ning and A. Engberts, "Program concept recognition and transformation." *IEEE Trans. Software Eng.*, vol. 18, no. 12, pp. 1065–1075, Dec. 1992.

[20] M. A. Linton, "Implementing relational views of programs" in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp.*, Practical Software Devel. Envir., May 1984. .

[21] N. H. Madhavji, "Compare: A collusion detector for pascal," *Technique et Science Informatiques*, vol. 4, no. 6, pp. 489–498, Nov. 1985.

[22] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.

[23] H. A. Muller, B. D. Corrie, and S. R. Tilley, "Spatial and visual representations of software structures: A model for reverse engineering," Tech. Rep. TR-74. 086, IBM Canada Ltd., Apr. 1992.

[24] S. Paul, "SCRUPLE: A re-engineer's tool for source code search," in *Proc. CAS Conf.*, IBM Canada Ltd. Lab., Centre for Adv. Studies, 1992.

[25] S. Paul, "Theory and design of source code query systems," Ph.D. thesis, Univ. of Michigan, 1994. In preparation.

[26] S. Paul and A. Prakash, "Source code retrieval using program patterns," in *Proc. 5th Int. Conf. Comput. Aided Software Eng.*, 1992, pp. 95–105.

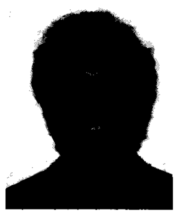
[27] D. Perry, "Software Interconnection Models," in *Proc. 9th Int. Conf. Software Eng.*, 1987.

[28] Reasoning Systems, Palo Alto, CA, *REFINE User's Guide*, 1989.

[29] R. Sedgewick, *Algorithms in C*. Reading, MA: Addison-Wesley, 1990, chapter 20.

[30] D. Swanberg, C. F. Shu, and R. Jain, "Knowledge guided parsing in video databases," in *Image and Video Processing Conf.; Symp. Electronic Imaging: Sci. & Tech.*, IS&T/SPIE, vol. 1908, San Jose, CA, February 1993, pp. 13–24.

[31] S. Westfold, " Pattern Matching in REFINE." Personal Communication, June 1993.



Santanu Paul received the B.Tech degree in computer science from the Indian Institute of Technology, Madras, in 1990 and the M.S. in computer science and engineering from the University of Michigan in 1992. He is a Ph.D. candidate at the University of Michigan, Ann Arbor. His thesis focuses on the design of algebraic languages to query source code. His research interests include databases, pattern matching, reverse engineering, and multimedia systems.

He was the recipient of an IBM Canada Graduate Research Fellowship during 1991–93. He is a student member of the IEEE Computer Society.

Atul Prakash received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi in 1982, and the M.S. and Ph.D. degrees in computer science from the University of California at Berkeley in 1984 and 1989, respectively.

Since 1989, he has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, where currently he is an Assistant Professor. His research interests include toolkits and architectures for supporting computer-supported cooperative work, support for reengineering of software, and parallel simulation.

Dr. Prakash is a member of the ACM and the IEEE Computer Society.