Program Concept Recognition

Wojtek Kozaczynski, Jim Ning, and Tom Sarver
Center for Strategic Technology Research, Andersen Consulting
100 South Wacker Drive, Chicago, Illinois 60606, U.S.A.

Abstract
Program understanding can be greatly assisted by automating the recognition of abstract concepts present in the program code. This paper describes an approach to automated concept recognition and its implementation. In the approach, we use a concept model and a library of concept recognition rules to describe what the concepts are and how to recognize them from lower-level concepts. Programming language knowledge as well as domain knowledge are both used to aid the recognition of abstract concepts.

1. Program Understanding
Program understanding, the understanding of program source code, is essential to many software engineering and re-engineering activities such as maintenance, debugging, verification, renovation, migration, and design recovery.

Many tools have been developed to facilitate program understanding. With very few exceptions, however, these tools are little more than browsers that present different text and graphical views of the code. For example, VIFOR [Raj90] graphically displays the calling and cross reference relations in FORTRAN programs. ESW, a COBOL re-engineering toolset developed by VIASOFT, extracts structure charts from PERFORM statements and produces condensed source listings to highlight "structurally significant" parts of programs. The program understanding tool prototyped at IBM [Cle88] supports cross referencing, calling relation, and control flow and data flow presentations. More recently, LaSSIE [DBSB91] demonstrated the need for capturing architectural and conceptual information (in addition to code-level information) to support program understanding. LaSSIE provides a classification-based representation and retrieval framework for the user to encode, browse, and query the information about high-level concepts; but these concepts must be obtained and encoded manually.

Browsing-based tools may assist the user in exploring hidden properties of programs. But it remains the user's responsibility to reason about the meaning of the programs. Moreover, as the user builds understanding of individual concepts, browsing tools cannot help him/her generalize the knowledge to automate the understanding of similar concepts in the future.

In this paper, we present an approach to software understanding based on an automatic identification of abstract concepts in the code. The following section gives a definition of concept recognition and discusses how concepts are recognized. Then, in Section 3, we introduce a language for specifying abstract concepts. In Section 4, we describe how our prototype concept recognition system uses the concept specifications to recognize abstract concepts. The last section summarizes the approach and discusses the related work.

2. Concept Recognition
What do we mean when we talk about understanding a program? Syntactically, a program is a sequence of text strings. But semantically, it contains many types of concepts, generally including language concepts and abstract concepts. Language concepts are variables, declarations, modules, statements, and etc. that are defined by the coding language. Abstract concepts represent language-independent ideas of computation and problem solving methods. Abstract concepts can be further classified into:

- programming concepts that include general coding strategies, data structures, and algorithms;
- architectural concepts that are associated with interfaces to architectural components such as operating systems, transaction monitors, networks, databases, etc.; and
- domain concepts that are application or business logic functions implemented in the code.
class MOVE-STATEMENT isa VERB-STATEMENT
SOURCE-DATA-ITEM : DATA-ITEM;
TARGET-IDENTIFIERS : seq(IDENTIFIER)

MOVE-STATEMENT ::= "MOVE" {("CORR" | "CORRESPONDING")} SOURCE-DATA-ITEM "TO" TARGET-IDENTIFIERS+

The class definition specifies that the MOVE-STATEMENT be a subclass of the VERB-STATEMENT. It has two attributes: (1) the SOURCE-DATA-ITEM which holds a concept of class DATA-ITEM, and (2) the TARGET-IDENTIFIERS which holds a sequence of IDENTIFIER concepts. The grammar rule specifies how the MOVE-STATEMENT concepts can be recognized from keywords (MOVE, CORR, etc.) and DATA-ITEM and IDENTIFIER concepts. The language concepts VERB-STATEMENT, DATA-ITEM, and IDENTIFIER as well as the rules specifying how to recognize them are also part of the COBOL language model and its grammar. Figure 1 shows a part of an AST with a MOVE-STATEMENT concept. Triangles in the figure represent omitted AST subtrees.

Automated recognition of abstract concepts is more difficult to achieve. In particular, abstract concepts may not be localized; they do not necessarily occupy consecutive sections of code. They usually interleave with each other and sometimes even share components. It is thus very expensive to search for these concepts lexically. Moreover, we know very little about what to search for. Theories about programming, architectural, and application domains are difficult to develop. It is unreasonable to assume these theories to be as complete and correct as theories about programming languages.

Thus, in automated recognition, partial recognition should be allowed. Typically, a partial recognition will not yield a tree structure (as an AST), but rather a forest of hierarchies rooted in the AST, as illustrated in Figure 2. Because of the inherent incompleteness of knowledge about abstract concepts, we may not be able to automatically recognize an entire program as an instance of one concept. We should, however, be able to recognize "islands" of concepts.

There is no magic behind automated concept recognition. To recognize language concepts, we must use knowledge expressed by language models and grammars. Similar knowledge is required in abstract concept recognition, which is the topic of the next section.


This section discusses how abstract concepts are specified in order to be recognized. Specifications of abstract concepts must address two questions: (1) what to recognize, and (2) how to recognize it.

Figure 1. AST representation of a MOVE-STATEMENT concept.

Figure 2. Recognizing abstract concepts (C₁ - C₆) from language concepts (L₁ - L₇).
The WHAT Question

A concept classification hierarchy, called a concept model, answers the what question. Like a language model, it establishes the ISA hierarchy between concept classes for which instances are expected in the target programs. In fact, this concept model subsumes language modules, as is shown in Figure 3.

Other than the ISA relationships, the concept model also defines attributes of concepts. These attributes represent descriptive features of concepts and are used to reference concepts by other concepts (this will become clearer later). Below are two examples:

BOOK-RECORD is a DOMAIN-CONCEPT
  RECORD-NAME : name of the book record
  TITLE : the title field
  AUTHOR : the author field
  PRICE : the price field
  IN-STOCK : the field that records # of copies in stock

ADD-NEW-COPIES is a UPDATE-BOOK
  ADD-TO-BOOK : name of the book record
  QUANTITY : new copies to be added

The meaning of the above concepts and attributes should be self-evident. ADD-NEW-COPIES, for example, represents the concept of adding new copies of a book to the book inventory. It has two attributes: ADD-TO-BOOK which holds the name of the record (BOOK-RECORD, another abstract concept) to be updated, and QUANTITY which holds the name of a program variable (a language concept) indicating the number of copies to be added.

As pointed out earlier, it is not necessary for a concept model to describe an extensive collection of abstract concepts to be useful. A typical scenario is to start with a small model and evolve it as the user gains deeper understanding of the programs under analysis.

The HOW Question

The how question cannot be answered with a purely syntactic approach, because the abstract concepts are not tightly related by syntactic structures. They are also connected by semantic relationships such as: control-flow relationships, data-flow relationships, calling relationships, and etc. For example, a READ-AND-PROCESS concept that sequentially reads and processes each record in a sequential file would require two of its component concepts (sub-concepts), the READ-RECORD and the PROCESS-RECORD concept, to follow each other on a control flow path. We call these semantic relationships and other requirements on the sub-concepts of abstract concepts constraints.

A specification of how to recognize an abstract concept must, therefore, contain information about: 1) components or sub-concepts of the concept, and 2) constraints on and among the sub-concepts. We encode this information in the form of concept recognition rules. For historical reasons [Ric81, Wil87, Is85, Let88, Nin89, Har91], we also use the term plan to refer to a recognition rule. The following is the structure of a plan:

    plan C consists of S_1, S_2, ..., S_m
    such that E_1, E_2, ..., E_n

218
where \( C \) describes the abstract concept to be recognized by
this plan; \( S_1, S_2, \ldots, S_n \) describe the sub-concepts; and \( E_1, E_2, \ldots, E_n \) are logical expressions (constraints). \( C \) as well as
\( S_1, S_2, \ldots, S_n \) are called pattern descriptions of concepts
and have the following form:

\[
\text{CONCEPT-NAME : (CONCEPT-CLASS(A_1 : V_1, A_2 : V_2, \ldots, A_k : V_k)}
\]

\text{CONCEPT-NAME} (optional) is a symbolic name given
to a concept. \text{CONCEPT-CLASS} denotes the concept class
and \( A_1, A_2, \ldots, A_k \) are the concept attributes.
\( V_1, V_2, \ldots, V_k \) are called patterns and may contain constants and pattern variables.
Constants require exact matching with objects in
the program. Variables, denoted by names starting with an
@ sign (e.g. @var), may match with arbitrary objects. Patterns are parsed into ASTs before they can be matched with
subtrees of the program AST to bind pattern variables. A
more complete description of pattern matching is beyond
the scope of this paper. Similar notions can be found in
[REF89].

Assume that \( C_1, C_2, \ldots, C_m \) are language and abstract
concepts already recognized in a program \( P \), as shown in
Figure 4a. Let us also define \( \sigma \), called a set of bindings.
Each binding in \( \sigma \) is a mapping from a pattern variable \( v \) to
a value \( l \), represented by a tuple \(<v, l>\). \( \sigma(v) = l \). More generally, we use \( \sigma(S) \) to describe the instantiation of all pattern variables in a pattern description of \( S \) to values defined
by \( \sigma \).

With these definitions, we say that a plan is applicable
to \( P \) at \( C_1, C_2, \ldots, C_m \) if there exists a \( \sigma \), such that:

1) \( \forall i: 1 \leq i \leq m \ C_i = \sigma(S_i) \); and
2) \( \forall \sigma(E_1) \& \sigma(E_2) \& \ldots \& \sigma(E_n) \) i.e. logical expressions
instantiated by \( \sigma \) are valid with respect to (or can be
derived from) \( P \), the syntax and semantics of the coding
language of \( P \).

When these two conditions are met, the plan will be
applied to \( P \) on top of \( C_1, C_2, \ldots, C_m \) to create a new concept
instance \( C_{m+1} = \sigma(C) \), producing Figure 4b.

The equivalence operator (=) in the first condition tests
the structure equivalence between two concepts. This is
implemented by comparing AST structures of corresponding
attributes of these two concepts. The second condition
specifies syntactic and semantic constraints. It is very diffi-
cult to derive the validity of these constraints from pro-
gramming language syntax and semantics using theorem
proving-based methods. To simplify the task, we have
developed a library of predicate functions to support the
evaluation of constraints. The necessary syntactic and
semantic knowledge has been encoded procedurally into
these functions. Below are some of the predefined func-
tions:

\text{CONTROL-FLOW}(A, B): \text{returns TRUE if there exists a}
control flow path from A to B.

\text{CONTROL-DEP}(A, B, C): \text{A is control dependent on the C}
branch of B, where B is normally a control predicate and
C is a boolean condition. Informally, this means that A is
on a control flow path from B to the end of the program
but there also exists an alternative path from B to the end
not passing A. A more formal definition can be found in
[FO87].

\text{DATA-DEP}(A, B, D): \text{A is data dependent on B with}
respect to D, which means that the value of D used at A
is the value set up at B. This is called reaching definition
problem in [ASU87].

\text{CALLS}(A, B) \text{ or CALLED-BY}(B, A): \text{there exists a CALL}
concept in A that invokes B.

\text{PARENT}(A, B) \text{ or PARENT-OF}(A, B): \text{B is a parent node}
of A or A is a sub-concept (part) of B.

\text{SUBSTRING}(A, B): \text{string A is a substring in B.}

\text{REAL}(A): \text{A is of class/type REAL.}

Notice, that a pattern description of a sub-concept may
refer to an abstract concept. If \( S_i \) is such a description, for
example, but there is no \( C_i \) already recognized in \( P \) such
that the above two conditions can be met, then plan(s) for
\( C_1 \) will be triggered to recognize its instances.

\text{Figure 4. Program concept recognition.}
Let us use a few concrete examples to illustrate the introduced ideas, as shown in Figure 5. Consider how to specify concepts that concern adding new copies of a book into a book inventory system. The plan for the top-level concept ADD-NEW-COPIES can be specified as plan P1.

In this plan, @rec, @new-copies, and etc., are pattern variables, and 1 (in the second sub-concept following OP2) is a constant. The plan states that the ADD-NEW-COPIES concept has four sub-concepts: book-rec, test, add-copies, and update. book-rec is an abstract, problem domain concept of type BOOK-RECORD. This record contains information about the book such as the amount of copies in stock (amount). The other three sub-concepts are programming concepts: test checks whether the current transaction is an add new copy transaction; add-copies adds the number of new copies to the existing amount; and update writes the record into the book inventory file. In addition, the constraints require add-copies to be control dependent on test’s TRUE branch and update to be data dependent on add-copies with respect to the data item @amount (normally, @amount is a data field in @rec). If instances of sub-concepts can be found in the program and if they satisfy the constraints, an instance of ADD-NEW-COPIES concept will be created:

\[
\sigma(\text{ADD-NEW-COPIES}(\text{ADD-TO-BOOK} : @\text{rec}, \text{QUANTITY} : @\text{new-copies}))
= \text{ADD-NEW-COPIES}(\text{ADD-TO-BOOK} : \sigma(\text{rec}), \text{QUANTITY} : \sigma(\text{new-copies}))
\]

If instances of BOOK-RECORD concept have not been recognized before, P1 will trigger plan P2. P2 says that the BOOK-RECORD concept can be recognized from a DATA-RECORD concept (a programming concept), provided that the fields of the data record meet certain naming and type constraints. Notice, that this is a heuristic rule because type

\[
P_1: \quad \text{plan ADD-NEW-COPIES(ADD-TO-BOOK : @\text{rec}, \text{QUANTITY} : @\text{new-copies})}
\]
\[
\text{consists of}
\begin{align*}
\text{book-rec} & : \text{BOOK-RECORD(RECORD-NAME : @\text{rec}, IN-STOCK : @amount)} \\
\text{test} & : \text{EQUAL-OP(OP1 : \text{@trans-type}, OP2 : 1)} \\
\text{add-copies} & : \text{ACCUMULATE(TO : @amount, FROM : @new-copies)} \\
\text{update} & : \text{WRITE-FILE(FROM : @\text{rec})}
\end{align*}
\]
\[
\text{such that}
\begin{align*}
\text{CONTROL-DEP}(\text{add-copies, test, TRUE}) \\
\text{DATA-DEP}(\text{update, add-copies, @amount})
\end{align*}
\]

\[
P_2: \quad \text{plan BOOK-RECORD(RECORD-NAME : @\text{rec-name}, TITLE : @the-title, AUTHOR : @the-author, PRICE : @the-price, IN-STOCK : @the-amount)}
\]
\[
\text{consists of}
\begin{align*}
\text{the-record} & : \text{DATA-RECORD(RECORD-NAME : @\text{rec-name}, ITS-FIELDS : (@the-title, @the-author, @the-price, @the-amount))}
\end{align*}
\]
\[
\text{such that}
\begin{align*}
\text{STRING(ITS-TYPE(@the-title))} \\
\text{SUBSTRING("TITLE", ITS-NAME(@the-title))} \\
\text{STRING(ITS-TYPE(@the-author))} \\
\text{SUBSTRING("AUTHOR", ITS-NAME(@the-author))} \\
\text{REAL(ITS-TYPE(@the-price))} \\
\text{SUBSTRING("PRICE", ITS-NAME(@the-price))} \\
\text{INTEGER(ITS-TYPE(@the-amount))} \\
\text{SUBSTRING("IN-STOCK", ITS-NAME(@the-amount))}
\end{align*}
\]

\[
P_3: \quad \text{plan DATA-RECORD(RECORD-NAME : @\text{rec-name}, ITS-FIELDS : @data-fields)}
\]
\[
\text{consists of}
\begin{align*}
\text{record} & : \text{COBOL-GROUP-ITEM(ITS-NAME : @\text{rec-name}, ITS-FIELDS : @data-fields)}
\end{align*}
\]

\[
P_4: \quad \text{plan COBOL-GROUP-ITEM(ITS-NAME : @name1, ITS-FIELDS : @fields)}
\]
\[
\text{consists of}
\begin{align*}
\text{group-item} & : \text{DATA-DESCRIPTION-ENTRY(PATTERN : "@level1 @name1." )} \\
\text{fields} & : \text{set of DATA-DESCRIPTION-ENTRY(PATTERN : "@level2 @name2 PIC @pic." )}
\end{align*}
\]
\[
\text{such that}
\begin{align*}
\text{PARENT(fields, group-item)}
\end{align*}
\]

\[
P_5: \quad \text{plan DATA-RECORD(RECORD-NAME : @struct-name, ITS-FIELDS : @data-members)}
\]
\[
\text{consists of}
\begin{align*}
\text{record} & : \text{C-STRUCT(ITS-NAME : @struct-name, ITS-MEMBERS : @data-members)}
\end{align*}
\]

Figure 5. Example concept recognition rules/plans.
and keyword checking alone cannot ensure correct recognition. We should also point out, that it is not necessary to use all the attributes to reference a concept. For example, plan $P_2$ instantiates five attributes of the BOOK-RECORD concept but only two of them are actually used in $P_1$ by the sub-concept book-rec.

Programming concepts, such as DATA-RECORD, can be further specified in terms of language concepts, as in $P_3$ and $P_4$. DATA-DESCRIPTION-ENTRY in $P_4$ is a language concept class defined in our COBOL language model. A description for a language concept contains only one attribute: PATTERN, which is used to match with program AST subtrees. For example,

```plaintext
@level2 @name2 PIC @pic.
```

will match with the AST representation of the COBOL data description entry:

```plaintext
05 RETAIL-PRICE PIC 9(3)V9(2)
```

producing the following set of bindings:

```plaintext
\sigma = [\langle @level2, 05 \rangle, \\
\langle @name2, RETAIL-PRICE \rangle, \\
\langle @pic, 9(3)V9(2) \rangle].
```

While DATA-RECORD in $P_3$ is expressed by a COBOL language concept (COBOL-GROUP-ITEM), it could as well be expressed by a C concept (which would be necessary if we were to analyze C programs), as $P_5$ demonstrates. It is important to allow the ability to specify one concept (DATA-RECORD) with several plans (e.g. $P_3$ and $P_4$). This is a useful feature even when analyzing programs written in the same language — one functional concept may have very different implementations that require different plans for recognition.

4. Recognition of Abstract Concepts

In this section, we describe how our implementation of the concept recognition system works. Figure 6 illustrates the major components and processes of this system.

### Parsing and Semantic Analysis

Parsing translates the analyzed program and the plans into AST representations and stores them in a knowledge base for further analysis. After parsing, the program’s AST is augmented with additional information. First, a cross-reference analyzer establishes cross-reference links between constant, variable, procedure, and label declarations and their references. Then, a control flow analyzer creates an extended control flow graph that captures not only control flows among basic blocks [ASU87] but also calling relations among procedure modules. Using the control flow information, a data flow analyzer computes definition-use chains [ASU87]. Finally, a control dependence analyzer derives the post-domination and control-dependence relations [FOW87].

### Recognition

The recognition process builds abstract concepts on top of the program’s AST nodes (which represent language concepts). This is a top-down process. Each recognition rule/plan is parsed into a plan object. A plan object has a method, called RECOGNIZE, associated with it to recognize instances. When triggered, this method executes the following steps:

![Figure 6. A program concept recognition system.](image-url)
1) Trigger the overloaded RECOGNIZE methods associated with sub-concept pattern descriptions. These methods return sets of recognized instances of the sub-concepts (one set per sub-concept).

2) Select a permutation of sub-concept instances, one from each set. The selected instances cannot violate binding requirement, which says that the same variable defined in different sub-concepts must be bound to equivalent knowledge base objects: \( \forall \text{var}_1, \text{var}_2: \text{var}_1 = \text{var}_2 \implies \sigma(\text{var}_1) = \sigma(\text{var}_2) \). In plan (1) for example, variable \( \theta \text{ rec} \) in book-rec and update must match with the same identifier object representing a record name.

3) Test the concept constraints on the permutation of instances. In our current implementation, constraints are expressions in a C-like syntax. We have developed an expression parser and an interpreter to parse and evaluate constraint expressions. The predefined constraint evaluation functions are invoked by the interpreter during constraint evaluation.

4) If all the constraints are satisfied, create a new concept instance with attribute values set up according to the header of the plan. Repeat step 2 to select another permutation of instances.

Presentation

The user interface has two types of windows, the plan window and the program window. This is shown in Figure 7. These windows are syntax-directed browsers in the sense, that text objects in the windows are directly connected with AST nodes in the knowledge base. When the user clicks on a point in a window, the “nearest” syntactic object is highlighted in its entirety. Subsequent clicks cause higher level syntactic objects to be highlighted. This feature allows very precise selection of syntactic boundaries so that concept recognition can be applied with specific plans (rather than all the plans) on specific regions of code (rather than the entire program). The user may open several windows of the same type, since we allow several plan files (also called plan libraries) and programs be loaded into the knowledge base during one session.

The user interface in our system serves not only to present the recognized concepts but also to accept user queries to drive concept recognition process. From the plan window, the user can select a plan of an abstract or a language concept and query for its instances. If there are already instances associated with the selected plan, the system simply presents them to the user. Otherwise, the RECOGNIZE method associated with the plan is triggered to recognize the instances. Similarly, from the program window, the user may select a region of code and query for concept instances present in the region. Again, this may trigger a concept recognition session if concepts in the given region have not been searched for before.

Once a set of concept instances has been retrieved, a list box is used to present them (see middle of Figure 7). The user can select one or more instances in the box. When an instance gets selected, the plan that contributes to its recognition is highlighted in the plan window. In addition, the code segments which make up the instance are highlighted in the program window. The user may use scrolling functions to display all the highlighted regions in the windows.

This concept browsing helps the user to understand how concepts are recognized (i.e. by which plans) as well as where they are located in the code.

Performance

It is not difficult to determine that the most expensive operation in concept recognition is the permutation over sub-concept instance sets (step 2 in the RECOGNIZE method). The complexity of this operation is:

\[
\prod_{i=1}^{n} |S_i|
\]

where \( n \) is the number of sub-concepts in the plan and \( |S_i| \) is the number of instances that match with sub-concept description \( S_i \).

Preliminary tests of our concept recognition system have shown that depending on the complexity of a plan, it takes from several seconds to a minute for the current implementation to recognize all the instances of a concept in a small program (a few hundred lines). This concept recognition system is a part of COBOL/SRE [KLN91b], a software renovation environment designed for large COBOL systems. It is therefore an imperative to handle large programs. To meet this requirement, we have implemented a number of methods to improve the performance:

![Figure 7. Browsing of abstract concepts.](image-url)
Early constraint evaluation. We force the evaluation of the constraints during the recognition and aggregation (generating permutations) of sub-concept instances rather than after. Since variables in the constraints may not be fully bound at the time of evaluation, the constraint interpreter was extended to handle unknown values.

Reordering of sub-concept instance sets. Constraints may reference more variables bound by some instance sets than others. Instances from sets more relevant to constraints should be selected and tested first under the assumption that they have a higher likelihood to fail the constraints. Another way of reordering the selection of sub-concept instances is based on the size of the instance sets. Instances from smaller sets get selected earlier so that a larger search space can be pruned.

Grouping sub-concept instances based on variable bindings. In a sub-concept instance set, we group into (not necessary disjoint) subsets instances which cause same pattern variables to bind with same program objects. Assume that adding an instance from such a subset to an incomplete permutation causes violation of the binding requirement or the constraints because of the common-value variable. Then, no other instances from the same subset would satisfy the binding and constraints, and therefore, the entire subset can be ignored.

The above enhancement should greatly improve the performance of concept recognition, although we have not conducted any extensive testing to support this claim.

5. Summary and Related Work

The approach to the recognition of language and abstract concepts described in this paper can be summarized as follows:

An automated approach. It attacks a difficult problem: automated program understanding. Despite the emphasis on automation, our approach also provides a user interface to support browsing and querying of abstract concepts.

A knowledge-based approach. Several types of knowledge are used in concept recognition. Syntactic knowledge about programming languages (language models) is used to generate language parsers that recognize language concepts. Semantic knowledge is used to compute the data flow, control flow, and dependency information to support constraint evaluation. Concept classification knowledge (concept model) is used to establish abstract concept classes for recognition. Finally, recognition knowledge (plans) is used to find concept instances.

An object-oriented approach. We use an object-oriented environment (ProKappa [Pro91]) to implement the concept recognition system. All language and abstract concepts are represented internally as knowledge base objects. Plans, as well as sub-concepts specified in plans, are also objects and have methods associated with them for recognizing concept instances.

A pattern matching-based approach. When a sub-concept in a plan attempts to find its instances, the corresponding attribute values are compared by pattern matching, which is a unification of ASTs of the attribute values.

While industrial-strength program understanding tools do not exist, a number of attempts have been made in the research community to find effective solutions to the problem. The directions of research have been greatly influenced by the ideas behind the Programmer's Apprentice project [RW88]. The notion of plans was first introduced formally in [Ric81] as a representation of control and data flow information about programs. Later on, [Wil87, RW90] developed a method of identifying algorithmic and data structure fragments by flow graph parsing [Bro84].

CPU (Cognitive Program Understnder, [Let88]) is a more representative program understanding tool. It takes a transformation-based approach to reverse-synthesize specifications from code. Plans and other standardization/normalization rules serve as correctness-preserving transformation rules to repeatedly rewrite programs into semantically equivalent but more abstract forms.

UNPROG [Har91], another program understnder, is a practical attempt to program understanding. It recognizes control concepts (a subset of programming concepts, such as iteration, read-and-process, bounded-linear-search, etc.) in COBOL programs based on data flow and control flow graph matching.

TAO (The Adaptive Observer, [BHW89]), a part of the DESIRE (DESign REcovery, [Big89]) system, takes a very different approach. It uses informal information, such as keywords embedded in comments, identifier names, and design documents, rather than formal information (syntax, semantics of programs) to search for abstract concepts. None of these efforts, however, have been pushed far enough to demonstrate practical utility.

Most of the above approaches to program understanding use the bottom-up derivation of high-level concepts from low-level ones. On the other hand, PROUST [JS85, Joh86] and DUDU (Debugging Using Device Understanding, [All90]) exemplify the top-down strategy of analysis.
They match functional goals against program pieces, using programming plans to fill in the missing details. DUDU, in addition, uses a weak theorem prover to handle situations when matching cannot be completed. The top-down approach was designed to deal with programs with bugs. The result of matching and theorem proving may provide explanations of the program errors. However, PROUST and DUDU were not intended to be general purpose program understanders. These tools require the user to specify top-level functional goals and they only answer how the goals are implemented in the code. They cannot derive what these goals are for arbitrary programs.

The work reported in this paper borrows ideas from many of the above approaches. It is also a continuation of the research that we have conducted in the past few years. PAT (a Program Analysis Tool, [HN89]) is also a pattern matching-based concept recognizer. PAT uses an interval logic to express semantic information such as control flow dependencies among sub-concept in order to facilitate computation and reasoning of abstract concepts. The BAL/SRW (Basic Assembler Language Software Re-engineering Workbench, [HKN91a]) is a knowledge-based re-engineering workbench for recovering designs from IBM System/370 assembler programs. Its automated code pattern understanding component, called ACPU, relates consecutive lines of assembler code with their functional goals by the use of plans, also defined in the form of pattern-matching rules. This environment has been successfully used in a number of commercial engagements.

There are many research issues yet to be solved. An obvious question, for example, is how to merge program understanding work with other related work such as software reuse, domain modeling, and other mainstream KBSE research. While exploration of these issues is potentially very interesting, we are also limited by many practical constraints. To name a few:

Ease of use. More often than not, knowledge engineering is the bottleneck in the application of knowledge-based techniques. Thus, a concept specification language should provide an intuitive and natural way for an average user to specify concept recognition knowledge. Otherwise, the user will prefer understanding the programs directly rather than taking the pain to specify plans.

Scalability. This is a question constantly raised in the AI and KBSE community. We have seen too many research prototypes which demonstrated well on "toy examples" but have failed to live up to the real-life expectations. Program understanding is a hard problem. It forces the developers to trade formal features, such as correctness and completeness, for practical goals, such as performance and flexibility, in order to produce industrial-strength tools.

Generality. We are talking about two levels of generality. At the programming language level, the techniques and methodology behind a program understanding system must be general enough to be applied to programs written in different languages. At the problem domain level, such a system should be general enough to analyze a large number of classes of programs. A "brittle" system (system that has a very narrow application domain) is of little use in this area because the goal is to understand unknown programs rather than to analyze same types of programs over and over again.

These practical considerations have been reflected in the design of our concept recognition system.

Currently, we are building re-engineering applications, such as program transformation [EKN91], segmentation, and concept browsing tools, on top of the recognition engine. They will increase the practical utility of the recognition system. In particular, we are planning to develop a tool that will help the migration of traditional CICS-based COBOL systems to distributed Client/Server architectures.

References


225