Program Representation and Behavioural Matching For Localizing Similar Code Fragments

Kostas Kontogiannis
McGill University
3480 University St., Room 318, Montréal, Canada H3A 2A7

Abstract

Reverse engineering focuses on the development of tools and techniques for understanding unfamiliar code. The main objective in design recovery is to understand program behavior. In order to understand the behavioral aspects of a program, concepts of language semantics and flow analysis can be used. In this paper we consider a program representation method in which communication of a code fragment with the rest of the system represents its behavior. Code fragments are viewed as objects capable of using resources and updating variables. Program similarity between simple code fragments can be proven in terms of the structure and the information residing in the nodes of program description trees. Program description trees are labeled trees in which nodes represent either code fragments or actions encoding the communication of a code fragment. Recognition and control algorithms for plan instance localization and recognition are investigated in order to assist partial design recovery when no complete recognition is possible. Finally, we discuss the underlying "Goal-Question-Analysis-Action" strategy for modeling the design recovery.

1 Introduction

Reverse engineering is concerned with the development of tools and techniques for understanding unfamiliar code, so that system maintenance can be facilitated [3]. A fundamental task of software maintenance and reverse engineering, is the design recovery. The term design recovery specifies the process in which a program is analyzed in order to be understood as a whole with respect to functional specifications, input parameters, expected output, performance, as well as on the software and hardware environment in which the program runs. The primary formal input is source code which is represented usually as a collection of text files. These files may contain a number of additional informal information on the functionality of the program such as, comments, print statements, other I/O commands, indentation, and meaningful variable names. Informal information though, is usually limited for the task of understanding complex features. These complex program features, have to be discovered using other techniques which operate on the level of abstract syntax trees (AST) or other abstract program representation schemes. The objective is to discover features such as the organization of program structure (how procedures or submodules are organized), the run-time functionality of the program (how and in what order modules are invoked), parameter passing, aliases, side effects, and meaningful system abstractions.

In the past few years several research groups have focused their efforts on the development of tools and techniques for program understanding and program restructuring. The major research issues involve the development of formalisms to represent program structure, control and data flow, and to visualize program execution. Design recovery for reverse engineering has many aspects. It may focus on generic code features (e.g., control flow, data flow) or specific design attributes in the code (e.g., memory use, uninitialized variables, value ranges, localization of specific algorithms). Each of these points of investigation must be addressed differently. The analysis and representation methods we want to investigate include

1. A code representation formalism to represent both structural and behavioral code features
2. A comparison algorithm to compare source code representations with programming plans that represent commonly used algorithms and programming structures.

*This work is funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto). Parts of this paper appear at "The Development of a Partial Design Recovery Environment for Legacy Systems" paper of CASCON ’93 Proceedings.
3. A top-level control strategy to select program parts for comparison with different programming plans in order to achieve plan instance localization.

Program representation is a key aspect for design recovery as it serves as the basis for any subsequent analysis chosen. Some of the most common program representation methods include (a) Abstract Syntax Trees [12], [5]; (b) Prolog rules [2]; (c) code and concept objects [5]; (d) code action frames [16]; (e) attributed data flow graphs [19]; (f) control and data flow graphs [6]; and (g) lambda calculus [11]. Most of these approaches represent and refer to the structural properties of a program. However, the main objectives in design recovery is to understand program behavior and, for this purpose, concepts from language semantics and concurrency can be used. Examples of contributions from language semantics are (a) denotational semantics, [17, 18] and (b) operational semantics [7] [15]. Concurrency has contributed towards program behavior analysis with the development of (a) Calculus for Communicating Systems (CCS) due to Milner [13] and (b) Communicating Sequential Processes (CSP) due to Hoare [8].

Denotational and operational semantics have been used with great effect for the mathematical understanding of the dynamic behavior of programs. CCS and CSP focus more on the communication aspects of processes and have been used successfully for the analysis of communication protocols and distributed systems.

The program representation scheme we consider is based on Program Description Trees (PDT). Program Description Trees are binary labeled trees which capture (a) basic control flow of a code fragment (e.g., branching, sequencing) and (b) communication of a code fragment with the rest of the system. Program Description Trees are created by using transformations applied to the Abstract Syntax Tree of a program. The behavioral definitions for each language construct are applied to the corresponding Abstract Syntax Tree node, thus building a Program Description Tree for this construct.

Once the representation scheme of the basic components of a program has been built, comparison methods to relate plan representations and source code fragments are applied. Comparison algorithms depend heavily on the program and on the selected representation method and do not always involve simple pattern matching. In this paper we consider a comparison algorithm which relates Program Description Trees to their structure and to the way code fragments represented as nodes in the tree communicate with the rest of the system.

2 PDT Architecture and Semantics

Program Description Trees are labeled binary trees which represent control flow (branching) and communication (read/write operations). Nodes represent code fragments or sequences of their related read/write actions. Labelled arcs represent mappings from code fragments to other code fragments or from code fragments to sequences of read/write actions. A Program Description Tree is built using a set of transformation functions applied to the corresponding Abstract Syntax Tree node.

For each syntactic entity of the language there is a transformation which maps the corresponding Abstract Syntax subtree to a PDT. Currently we have considered transformations for the basic syntactic entities of the PL/AS language 1. Specifically we have used Program Description Trees to define the behavior of (a) declarations (e.g., character, string, integer, binary); (b) references; (c) expressions (e.g., arithmetic add, subtract, multiply, divide, abs function, mod function, etc.); (d) assignment statements; (e) If-Then-Else statements; (f) GoTo statements; (g) While statements, and; (h) Select statements.

There are four basic labels in the arcs of a PDT. The Operation-Sequence label represents the communication of the parent node with the rest of the system in terms of a sequence of read/write operations. The Next-Control label indicates the continuation of the control flow after the execution of the sequence of read/write operations as specified by the Operation-Sequence. The two labels Branch1 and Branch2 are used to indicate branching and choice for different control paths. There is a special label called Iterate-To-Root but it is considered more as an attribute of the specific node Iterate rather than as a PDT arc label. The purpose of the Iterate-To-Root attribute is to encode iterations without creating graphs. An example of a PDT is shown in Fig. 1.

Program Description Trees (PDT) are implemented in an object-oriented environment [10], in which a domain model defines a hierarchy of classes and subclasses. Objects that are created during parsing or during the application of a transformation function become instances of these predefined classes and subclasses. Labels are also defined in the domain model.

---

1 PL/AS is a copyright of IBM Corp.
as mappings between objects. The environment allows for a number of tree operations, such as comparing two trees, selecting subtrees, and traversing the trees in preorder/postorder.

Currently there are two domain models in the system: one to describe the hierarchy of the objects in the Abstract Syntax Tree and one to describe the hierarchy of the objects in the corresponding Program Description Tree.

Complex PDTs are created by composing simpler PDTs. The composition algorithm operates on both the structure of the tree and the sequence of the performed read/write operations. Composition of PDTs is based on the CCS formalism of communicating processes [13]. The semantics of the composition affecting the structure of the PDTs are illustrated in Fig. 2.

The first rule shows the composition of two PDTs having both alternative branches. The result is a PDT in which the leaves are all the possible ways of composing the alternative branches of the original trees.

The second rule shows the composition of a PDT with an operation-sequence (os) arc and a next-control (nc) arc, with a PDT having alternative branches (b1 and b2). The result is a PDT which has two alternative branches. Each branch contains the operation-sequence (ie.\([a_1, a_2, \ldots, a_n]\) of the first tree and as next-control the composition of the next-control of the first tree with the corresponding next-control of the second tree.

The third rule describes the composition of two PDTs having operation-sequence and next-control arcs. The result is a PDT with an operation-sequence and a next-control arc. The operation sequence of the resulting PDT is computed by trying the operations in the first tree in sequence up to the point where it finds a complementary operation in the sequence of the second tree, when both operations are consumed and the index in both sequences advances by one. Finally, the Next-Control arc of the resulting tree is the composition of the Next-Control nodes of the two composed trees. Note that two operations are complementary if they have the same name and one of them is over-lined. Finally, the fourth rule simplifies a PDT by collapsing intermediate Next-Control nodes, and appending their corresponding operation-sequence to the operation-sequence of the node above. The meaning of complementary actions is that they serve as communication points for creating abstractions and linking operation sequences. The semantics of the formalism from which PDTs were built can be found in [13].

An example of an If-Then-Else statement represented as a Program Description Tree is illustrated in Fig. 3.
The If-Then-Else statement gives rise to a different branch according to what Cond evaluates to (Fig. 3(4)). The first branch consists of a subtree created by the composition of the PDTs corresponding to the Cond and the Then part of the If-Then-Else statement. The second branch consists of a subtree created by the composition of the Cond and the Else part of the statement of the If-Then-Else statement. The Cond is considered as an Expression in the domain model and its corresponding PDT is shown on the leftmost part of Fig. 3(3). The corresponding PDT for an assignment statement is shown in the rightmost part of Fig. 3(3). The right-hand sides of the assignments of the Then and Else parts are identifier references that are considered Expressions as well in the domain model. Their corresponding PDTs are shown in Fig. 3(1), Fig. 3(2) and in the leftmost part of Fig. 3(3). Using the composition rules defined above the complete PDT for the code fragment If (Cond) then $w = y$ else $w = z$ is shown in Fig. 3(5). The composition process applies the following rules (1) The general structure of an If-Then-Else statement is selected; (2) The Cond which is an Expression is transformed to its corresponding PDT; (3) The identifier reference $y$, which is considered an expression is transformed as a PDT; (4) The PDT for the assignment $w = y$ is created (note that the operation-sequence for this tree is $\text{read}_y(\text{val}), \text{write}_w(\text{val})$); (5) The Empty node and the Done node are composed creating a Done node; (6) The Cond PDT with the precondition it evaluates to True, and the Assignment PDT are composed creating the Then part of the final tree; (7) The identifier reference $z$, which is considered an expression, is transformed as a PDT; (8) The PDT for the assignment $w = z$ is created (note that the operation-sequence for this tree is $\text{read}_z(\text{val}), \text{write}_w(\text{val})$); (9) The Empty node and the Done node are composed creating a Done node; (10) The Cond PDT with the precondition it evaluates to False, and the Assignment PDT are composed creating the Else part of the final tree;

3 Comparison Methods

Comparison methods are used to perform simple plan instance recognition by proving equivalence or showing a partial order relationship between two simple programming plans. A simple programming plan
is a plan that has no other plans interleaved. In our approach such simple plan instance recognition is performed by defining relations on Program Description Trees.

Plans can be described at different levels of abstraction. The most abstract level is the one corresponding to the most concise description of a very complex system. With new representations, plan fragments at higher levels of abstraction could be detected and described. The most common plan-code comparison methods include:

- simple pattern matching,
- similarity metrics,
- graph matching, and
- body structure.

Some systems (eg. PROUST) [9], [5] match syntax trees with syntax tree templates. A plan matches a program statement if its unified template matches the statement's syntax tree and its constraints and subgoals are satisfied. SCRUPLE matches code fragments by specifying high-level regular expression patterns. [15] [14] compares student and reference functions by applying a heuristic similarity measure. In CPU [11], programs are represented as lambda calculus expressions and procedural plans. Comparisons in CPU are performed by applying a unification and matching algorithm on lambda calculus expressions. In UNPROG [6], program control flow graphs and data flow relations are compared with the programming plan's control flow graph and data flow relations. Quilici [16] matches frame schema representations of C code and if they structurally match then data flow graphs are compared too. GRASP [19] uses attributed data flow subgraphs to represent programs and programming plans. Comparisons are performed by matching subgraphs and by checking constraints involving control dependencies and other program attributes.

Our comparison algorithm relates two PDTs according to their structure and the sequences of read/write operations allowed at each node of the tree. Specifically we define similarity in terms of the $T_1$ performs-as $T_2$ relation ($T_1 \pi \ T_2$) 2 and $T$ executes-sequence $s$ relation ($e T(s)$) which are defined as follows:

(A) $T$ executes-sequence $s$ :

---

2The convention is that a PDT is named after its root node
• if \( os(T) = 0 \), then \( e_T(s) = T \)
• if \( os(T) = s, s \neq 0 \), then \( e_T(s) = nc(T) \)
• if \( os(T) = s \), and \( s1 \) is a prefix of \( s \), then \( e_T(s1) = T \) with \( os(T) = os(T) - s1 \)

\[ \text{(B) } T_1 \text{ performs-as } T_2 \]

• The empty PDT \( \emptyset \) performs-as an empty PDT \( \emptyset : \emptyset \pi \emptyset \)
• if \( T_1, T_2 \) are PDTs with roots \( T_1, T_2 \) respectively, then \( T_1 \pi T_2 \) if

- \( b1(T_1) \pi b1(T_2) \)
  (if \( b1(T_1) \) and \( b1(T_2) \) are defined) or
- \( b1(T_1) \pi b2(T_2) \)
  (if \( b1(T_1) \) and \( b2(T_2) \) are defined) or
- \( b2(T_1) \pi b1(T_2) \)
  (if \( b2(T_1) \) and \( b1(T_2) \) are defined) or
- \( b2(T_1) \pi b2(T_2) \)
  (if \( b2(T_1) \) and \( b2(T_2) \) are defined) or
- \( T_1 \pi b1(T_2) \)
  (if \( b1(T_2) \) is defined and \( b1(T_1) \) not defined) or
- \( T_1 \pi b2(T_2) \)
  (if \( b2(T_2) \) is defined and \( b1(T_1) \) not defined)

- \( os(T_1) = s_1 \) and \( os(T_2) = s_2 \) and \( \exists s \) such that \( s \) is a prefix of \( s_1, s_2 \), and \( e_T(s) \pi e_T(s) \).

Two PDTs \( T_1, T_2 \) then can be defined as similar:

\[ \text{(C) } T_1 \text{ similar } T_2 \]

- \( T_1 \pi T_2 \) and \( T_2 \pi T_1 \).

This relation resembles the bisimulation relation defined within the framework of CCS in [13]. An example of two similar PDTs are shown in Fig. 4.

\[ \text{Figure 4: An example of similar Program Description Trees.} \]

The similar relation can be used to relate two code fragments that have the same functionality but completely different structure. For example, a While construct can be proven similar to an If-Then-Else construct when the latter is combined with a GoTo statement (Fig. 5). In this example the corresponding PDTs are similar since they are identical trees. Finally, another example is that a Select construct can be proven similar with a sequence of nested If-Then-Else constructs (Fig. 6).

The advantage of comparing PDTs instead of Abstract Syntax Trees is that no specific rules for defining similarities are needed. The only rules (transformations) needed are those defining the semantics and the functionality of the language constructs. Similarity can be proven automatically by defining relations on these semantic program representations. Moreover, code localization becomes syntax independent and matching focuses on functional and not on structural properties of the code. In general, showing program equivalence is not a decidable task but under relaxed conditions (e.g., no need to know if a program terminates), we can use the above technology to develop behavioral code localization algorithms. Currently a limited number of PL/AS language constructs have been denoted but the translation process has shown that functional pattern matching can be feasible when program similarity can be proven under relaxed conditions.

4 Top-Level Control

Program parts and programming plans represented at higher levels of abstraction are selected using a
Figure 5: Similarity on PDT for a While statement and an If-Then statement combined with a GoTo statement.

Program decomposition can be used to guide the selection process. Performance is best when decomposition produces program parts that correspond to plans in the library. Program decomposition can be performed \textit{a priori} before the selection process starts or in a dynamic way based on previous recognition results and the current needs of the application as the selection process is performed.

For the control strategy, we focus on investigating an \textit{island-driven opportunistic search} in which search subgoals are set around some well-recognized point in the program. The idea is to use this positively identified point as an anchor and then try to satisfy subgoals around it.

Islands are obtained by starting with design decisions that have been positively (or with high evidence) identified and are of particular semantic relevance and then proceeding outward, extending the analysis in both directions. Irrelevant and intermixed plans can be ruled out by allowing recognition gaps and partial recognition of plans.

As an example, consider intermixed plans or parts of plans that do not share data dependencies. Such plans can be distinguished irrelevant; the island-driven search will skip them and proceed towards what it considers the current goal to be.

Such an island-driven search starts by establishing a top-level goal which, in practice, is the satisfaction of some programming plan. The establishment of the top
level goal is achieved by examination of the properties of the well-identified program part and the properties of the plans in the plan base. The search continues by trying to prove the existence in the code of other parts of the plan as these exist in the generic plan. Throughout the process, incomplete and partial satisfaction of the plan properties causes the search to jump in an opportunistic way, trying to satisfy continuously whatever it can. Island-driven opportunistic search will not fail if the plan and the code representation do not match exactly. In this case the island-driven search will jump to another point with high likelihood of recognition and will start again from this point. The search will terminate when no further plan recognition can take place. Thus, even partially matched plans will be reported by such an algorithm. Failure occurs only when none of a plan's subparts can be positively recognized. The influence of constraints between nonadjacent fragments can be investigated in the case where other fragments are left as uninterpreted gaps. These concepts can be used to specify interactive tools that propose an interpretation of a fragment to the maintainer who may not be interested in every fragment or who may provide a personal interpretation of some uninterpreted fragments.

Finally, the nature and the dynamics of the process used for design recovery suggest a model that is based on a Goal-Question-Analysis-Action paradigm. This strategy focuses on the improvement of the software process by considering the specific project goals and environments. In such a way, project objectives and the domain-specific information in the development environment guide the selection of appropriate models, methods, and tools in the software process.

In this model, the maintainer operates within a framework of top-level goals. These goals provide the framework and the justification for a series of questions that must be addressed to satisfy the top-level goals. Specific goals give rise to specific questions which require the use of specific analysis strategies and tools. Once the strategy and tool have been selected, an action is recommended. The maintainer may require additional information and can set new goals, reapplying the whole model (Fig. 7).

5 Research Issues

Large legacy systems represent significant assets for the companies that use them. These systems have evolved over time and tend to require continuous maintenance. The definition of the objectives for a design recovery process is based both on a generic list of desired design attributes to be recovered (such as call graphs, data types, value ranges, control flow, data flow) and on domain-specific design attributes of a specific product or specific language.

Within the framework of this project, maintenance objectives were identified during a sequence of contextual interviews with the Program Understanding Group (PUG) at IBM's Centre for Advanced Studies. The top-level objectives of the Program Understanding Project (PUG) address problems related to

a) code correctness and;

b) performance enhancement.

The research issues which arise in such a framework and constitute the objectives of our work include:

- selection, incorporation, and use of an appropriate process model;
- development of code representation schemes and plan localization techniques that are appropriate for mechanical manipulation and allow for the analysis of structural and behavioral properties of the code; ram dependencies)
• normalization and correction of erroneous code (e.g., removing dead code, simplifying or removing redundant expressions, localizing erroneous communication points); and

• integration and use of different technologies (flow analysis, language semantics, pattern matching, search and control strategies) in a reverse engineering environment.

Existing Program Understanding systems attempt to recognize plan instances by comparing program representations against programming plans. A common theme in these approaches is to use a program representation formalism, a plan repository, a plan localization control strategy, and a comparison algorithm.

The major problems associated with the program understanding systems built so far can be summarized as follows:

• Repository completeness: it is not possible to encode and store all possible plans occurring in an application, and usually the ones encoded are trivial cases (e.g., sorting algorithms, list traversals).

• High complexity: most understanders perform well only in small and medium-sized (approx. 5000 lines) programs. In large programs complexity makes design recovery an extremely difficult and time-consuming process. Difficulties are caused by interleaved and scattered plans and syntactic or implementation variations.

• Structural and lexical matching: in most applications, plan-program similarity or equivalence is determined by testing structural and lexical information from both the code and the plan. This causes a problem when the code fragment contains other plans, irrelevant statements, or when the plan is scattered among different parts of the application, because the structural and lexical-based representations of the code and the plan cannot be matched. Moreover, the behaviour of a program can not be encoded and represented.

• Graph-based matching: not all approaches use program representations that are based on structural and lexical information. Several program understanders use specific graph-based formalisms which incorporate data and control flow. Graph grammars are used to perform abstractions and to perform plan instance recognition. The problem in these applications is that graph transformations are usually very expensive to compute and graph pattern matching algorithms can have high complexity. This imposes a serious problem when large programs are analyzed because their corresponding graph representation can be very large and complex.

The approach taken in this project focuses on minimizing the drawbacks of other existing approaches. Below, we examine the basic views that we have adopted and the points of difference with respect to existing approaches.

• Use of a development strategy. In our approach we use a strategy for performing a Reverse Engineering (Design Recovery) task. This strategy was inspired by the "Goal-Question-Metrics" (GQM) strategy described by Basili in [1]. Our strategy, the "Goal-Question-Analysis-Action" strategy, is task oriented, goal driven, and focuses on decomposing the task of design recovery into a number of interrelated but distinct subtasks. This can be justified in two ways. Firstly, it is a natural way to perform design recovery. In practice, human maintainers do not try to understand the whole program at once but, instead, they gather different pieces of information and then they relate them using their expertise and their programming skills. Secondly, maintainers gather this information in a goal-driven way. At the beginning, they establish an objective
(e.g., "find what this procedure returns and how it computes its returned values", "Is it a sorting algorithm?"), and then they gather information (e.g., "find where this variable is updated", "when does this loop terminate?"), which they believe is relevant for meeting the specific objective. Information gathering is not a random process but is guided by the experience and the programming skills of the maintainer. We believe that this experience is valuable and that a Reverse Engineering environment should give the programmer the flexibility to specify his own queries and provide him with the tools to gather the information he thinks is important and relevant. Thus, the question the maintainer asks is the key to selecting the right analysis tool and the required actions to be performed.

- **Use of behavioral program representation model.** This approach uses a program representation method which reveals the behavioral aspects of the represented program. Specifically, instead of using a representation method based only on structural and lexical properties of the program, we use information which is based on language semantics and focuses on how a computation is performed, on what is the effect of a computation after its termination. The advantage of this approach is twofold. Firstly, it is based on a well-established formalism which is supported by a solid mathematical theory (language semantics). Secondly, the mathematical theory supporting the model serves as a foundation for defining equivalence and partial order relations which can then be used to relate plans and programs. Most program understanding applications use program representations that incorporate control flow, data flow, and structural information. The drawbacks of these representation schemes are that plan-program equivalence has no mathematical foundation and that they are based on simplistic pattern-matching algorithms. The complexity of these formalisms is high, which makes the design recovery of realistically large programs difficult. Our approach, which uses language semantics, has not been investigated yet on complexity issues, but it definitely gives an advantage on plan-program matching, which can be achieved based not only on structural and syntactic properties, but also on semantic and behavioral properties of programs.

- **Use of an efficient and flexible plan localization strategy.** Plan localization is a key issue in most program understanders. Plan localization algorithms must cope with problems due to syntactic variations, interleaved plans, implementation variations, overlapping implementations, and unrecognizable code. Some of the most common plan localization algorithms used in existing applications are top-down goal agendas, depth-first search, best-first search, repeated traversals, and exhaustive search. The drawbacks in most applications are (a) the complexity issues and (b) failure to produce any results if perfect localization cannot be achieved. Our approach has the complexity of an island-driven parser [4] and has the advantage that if the plan localization algorithm fails to recognize a plan completely it still recognizes parts of it, so that partial plan-program localization can be performed. In most cases, then, the maintainer can use his experience to recognize the rest of the plan himself.

- **Use of an integrating environment.** As indicated above, the view adopted in this project favours programmer-assisted program understanding over automatic program understanding. Our approach considers plan localization only as a subgoal of design recovery. Plans should be instances of principles and not simply instances of abstracted code fragments. Principles could be high-level descriptions of concepts occurring in the code (something that we believe it is not feasible with the current technology) or practical properties of the code itself (e.g., data dependencies, control dependencies, preconditions, and postconditions) with which programmers are familiar. Within this framework, a programmer may set high level-objectives with abstracted concepts. In order to address these objectives, he examines practical properties of the program such as control and data flow, dependencies, partial correctness properties, communication points, and informal information. A Reverse Engineering environment should provide to the maintainers tools for addressing these practical questions and not rely exclusively on automatic plan recognition using a static plan library. This approach fits well with the Goal-Question-Analysis-Action" process model, and has not been incorporated in any of the existing program understanders.
6 Conclusion

This paper describes an approach for program and system representation using Program Description Trees (PDTs). The use of this representation for system understanding offers several advantages. Firstly, program behavior is compositional and is given in terms of the behavior of its subcomponents. Program representation in PDTs is achieved by denoting the programming language constructs as communicating agents and by uniquely defining their semantics in terms of their behaviors. Formalisms used to support such a program representation are derivation trees and process algebra equations as these are defined within the framework of CCS. Secondly, semantic similarities between Program Description Tree descriptions of programming plans and source code can be defined. Similarity can be based on the observable behavior of every process in terms of its communication with the rest of the system. Finally, it is a uniform, natural, and scalable approach since program statements, procedures, modules, programs, operating systems, and users are all viewed as communicating agents. Therefore there is no distinction between program and system understanding. Program fragments are viewed as a collection of communicating agents capable of performing specific actions. We investigate the idea of using more sophisticated equivalence relations and partial order relations to show semantic equivalences between programming plans and source code representations. Partial order relations can be used to describe properties of a program fragment when no perfect, behavioral matching between plans and source code descriptions is possible due to syntactic variations, implementation variations, noncontiguousness, and unrecognizable code.

The critical problem of the completeness of programming plans seems to be overwhelming for efficient plan based program understanding. The number of programming plans required to cover all possible, realistic behaviors that can occur - let alone determine them - imposes a serious limitation on automatic program understanding. An alternative solution could be incremental recognition in a goal-oriented reverse engineering environment driven by an efficient query system.

Reverse engineering an entire system is expensive and this effort should be goal oriented. Careful analysis of different maintenance metrics should impose restrictions and define goals for the reverse engineering process. Goals can be set by the maintainer and expressed as a series of queries. Queries can be used to locate code fragments satisfying a specific behavior without considering possible variations due to implementation and syntactic differences between code fragments. Such queries may seek code that updates a specific variable, interfaces with a given module, or performs a particular sequence of actions, implementing a specific algorithm. Thus, plan recognition becomes a goal-driven, user-assisted process.

This approach is being applied for analyzing and understanding PL/AS programs at IBM Toronto research laboratories using the Refine 4 programming environment.

About the Author

Kostas Kontogiannis is a Ph.D student at Computer Science Department, McGill University. Kostas has a B.Sc in Mathematics from University of Patras, Greece and an M.Sc in Computer Science and Artificial Intelligence from Katholieke Universiteit Leuven in Belgium. He is working in the area of software design recovery, pattern matching, and artificial intelligence. He can be reached at kostas@cs.mcgill.ca.

References


[5] Engberts, A., Kozaczynski, W., Ning, J. “Automating software maintenance by concept recognition-based program transformation,” IEEE Conference on Software Main-

---

4Refine is a trademark of Reasoning Systems Corp.


