The Program Understanding Problem: Analysis and A Heuristic Approach

Steven Woods

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada
sgwoods@logos.uwaterloo.ca

Qiang Yang

School of Computing Science
Simon Fraser University
Burnaby, British Columbia V5A 1S6
Canada
qyang@cs.sfu.ca

Abstract

Program understanding is the process of making sense of a complex source code. This process has been considered as computationally difficult and conceptually complex. So far, no formal complexity results have been presented, and conceptual models differ from one researcher to the next.

In this paper we formally prove that program understanding is NP-hard. Furthermore, we show that even a much simpler subproblem remains NP-hard. However, we do not despair by this result, but rather, offer an attractive problem-solving model for the program understanding problem. Our model is built on a framework for solving Constraint Satisfaction Problems, or CSPs, which are known to have interesting heuristic solutions. Specifically, we can represent and heuristically address previous and new heuristic approaches to the program understanding problem with both existing and specially designed constraint propagation and search algorithms.

1 Introduction

An expert attempts to understand the source code of a legacy program as part of the legacy maintenance task. Understanding can be thought of as at least the process of constructing mappings between existing knowledge and some perceived artifact. In the domain of maintaining legacy source, an expert would attempt to construct these mappings based upon a previous knowledge set which includes general knowledge about how programs are constructed and specific knowledge about typical program plans. We describe abstracted program plans as *templates*, both generic and domain specific. Given a particular store and representation of expert programming *knowledge*, and some *perceived* structure of a legacy artifact, one might describe the process of constructing a mapping between these as understanding. An understood or partially understood program is one which has had its structures and components at least partially contextualized in the scope of existing knowledge about programs and programming. Once constructed, an expert can, and typically does, use these mappings to infer other higher level and lower level goals for a source program. The mappings raise the level of abstraction of the legacy code representation from purely raw source level to include the more abstract level of the existing representational framework for expressing domain knowledge. This abstracted representation may subsequently be exploited as part of the process of at least the following tasks:

1. translating the program into the source code of another programming language,
2. recognizing errors in legacy code and assisting in debugging the code, or
3. replacing understood code portions with generic application code or calls to other code libraries.

There have been a variety of methods proposed which partially solve the program understanding problem, primarily as parts of a supposed *interactive assistant* or *maintenance toolset*[11, 13, 14, 5, 9, 10]. Each of these approaches attempts to integrate perceptions or recognitions of particular abstracted program plan templates into an overall understanding of the source in terms of a particularly configured library of pre-existing knowledge about how programs (in general or in a particular domain) are known to be structured. Past work has not provided a formal analysis of the complexity of the understanding problem, or presented an understanding framework shown to be general enough framework to include most of the previous approaches.

In this paper, we present two results.

- First, we show that the program understanding problem is NP-hard. This result provides a formal justification for researchers to look for heuristic methods for solving the problem.

- Second, we present a CSP (constraint-based) model for program understanding. This model allows much of the previous work to be cast under a unifying framework. We illustrate this model with associated algorithms and examples.

2 Previous Approaches

In Artificial Intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition [4, 1]. These programs have been applied mostly to *toy domains* (such as the cooking domain), involving small knowledge bases and a small amount of search. Woods et al[16] describe in more detail how current approaches to program understanding fundamentally differ with earlier and some current plan recognition methodologies.

Recently, researchers have adopted a more direct approach to program understanding. An explicit library of programming plan templates and concepts is constructed, and various top-down and bottom-up search strategies are utilized to implement the mapping process. Kozaczynski and Ning[5] describe a method of automatically recognizing abstract concepts in source code. Given a library of concepts and a set of rules for how to recognize the higher-level concepts from lower-level language concepts, the search in their `Concept Recognizer` is controlled in what is essentially a top-down, library-driven manner. Rich and Waters[11] headed the Programmer's Apprentice project which focused on the development of a demonstration system (Knowledge-Based Editor in Emacs or `KBEmacs`) with the ability to assist a programmer in analyzing, creating, changing, specifying and verifying software systems. In addition, Rich and Waters[11, pp. 171-188] describe a cliché recognizer called `Recognize`, based in `KBEmacs`. Understanding tools rely on the existence of structured legacy source. As part of the effort at providing this structure for maintenance engineering, Muller et al[8] are involved in the construction of `Rigi`, a system for analyzing software systems which includes visual representations of data and control-flow structures in a code for the identification of subsystems and hierarchies of structure in the code. Along this line, Devanbu and Eaves[2] have constructed `Gen++`, a tool which generates tools for analysis of C++ code. Specifically, `Gen++` can generate tools which in turn generate annotated abstract syntax trees (ASTs) of C++ code showing control and data-flows.

In subsequent sections we review two major themes representative of the endeavor to create understanding tools. In Figure 1 a subset of expert knowledge about a particular application domain is represented in a fragment of a hierarchical library of program templates. One possible mapping is shown between a plan template from the library and a specific legacy source fragment, in this case a single source statement. The existence of such a mapping essentially *explains* the presence of the low-level source statement at a higher level of abstraction, as an instance of the plan template **copy-character** specified in the library.

There has been a rich tradition of heuristic approaches to program understanding. One recent approach by Quilici[9, 10], a derivative of earlier work by Kozaczynski and Ning[5], is based on a construction of an explicit library of programming plan templates, complete with an indexing ability, which can quickly associate a particular recognized source code fragment with program plan templates in the knowledge base. In this "code-driven" fashion, a combination of top-down and bottom-up search strategies is utilized to implement the matching process. With his `DECODE` system, Quilici demonstrated how simple C programs could be translated to C++ programs. This approach marks one of the first cognitively motivated[1] attempts to program understanding using a hierarchical library of program plans.

Program plans, such as those embedded in Abstract Data Types (ADTs), are organized hierarchically in a library as shown in Figure 1. Legacy source code preprocessed as an annotated AST is mapped to the plan library through the use of indices. Indices are pre-defined "keys" or pointers in the plan library mapped to key instances in the legacy source. Index tests indicate when it is appropriate to *specialize* or to attempt to *infer* the existence of other plans according to a set of pre-defined conditions. As an example of specialization, consider Figure 1 in which the program plan **initialize-string** is specialized to **builtin-char*-copy** when a direct string assignment is observed in the source code. An example of an inference test is also shown in Figure 1, where the existence of **loop-initialize-string** is inferred when an instance of **loop-through-character-array** is "near" a related instance of **copy-character** in the source code.

In a different approach, Wills[11, 13, 14] models stereotypical program or data structures (*clichés*) as a type of flow-graph grammar and parses[2] legacy source represented as a flow graph. Each successful partial parse represents a one explanation of part of the source program.

3 Complexity Analysis

Our survey of the approaches to program understanding has resulted in the following model. One is given a source program to understand in terms of a library of program plan templates. From these, one is to compose a solution in

[1] Quilici's work has included observation of the behaviour of student programmers in manipulating legacy examples.

[2] Wills notes that although the parsing problem is NP-complete in general, experience suggests that attribute constraint checking significantly prunes the search space in practice.

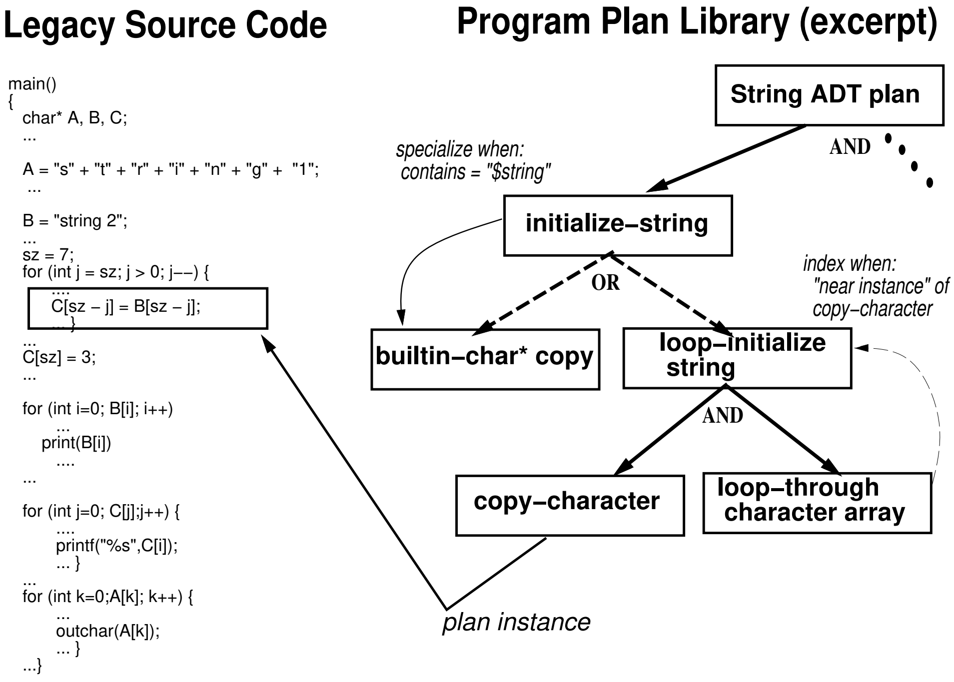

Figure 1: Conceptualizing source with a plan library.

the form of a mapping from portions of the source code to part of the plan library. In this section, we prove that this problem is intractable.

Simple Program Understanding Problem

Our strategy will be to simplify the problem. Consider the following **Simple Program Understanding (SPU)** problem, depicted in Figure 2. We are given the following:

- The source code consists of a collection B of program blocks $B_i, i = 1, 2, ..., m$. These blocks can be viewed in terms of a corresponding graph $P = (B, D)$ where D is the set of edges of the graph $D_k, k = 1, 2, ..., n$, such that an edge $D_{i,j}$ exists between nodes B_i and B_j if and only if a data-flow exists between the program blocks.

- We are also given a library of program plan templates represented as a graph $L = (T, C)$ where T, the set of templates $T_o, o = 1, 2, ..., t$ in L are related to one another through data-flow relationships. These relations are specified by a set C of edges, such that an edge $C_{k,l}$ exists between templates T_k and T_l if and only if a data-flow possibly exists between them.

Given the above structure, the SPU problem is to determine if a correspondence exists from program blocks to a subset of templates. The correspondence is of the form of a mapping between templates and program blocks, and between their data-flow relationships. As an example, in Figure 2, the following correspondence gives rise to an understanding of the example program:

$$\begin{aligned} B_1 &\iff T_3 \\ B_2 &\iff T_6 \\ B_3 &\iff T_4 \\ B_4 &\iff T_5 \end{aligned}$$

We contend that the SPU problem is representative of many program understanding tasks. In Kozaczynski and Ning's approach in the `Concept Recognizer` system, program plans which consist of components and constraints abstracted away from a particular implementation language or method are utilized. Quilici extends these plans with the provision of indices (memory) that control the selection of candidate plans more selectively than in `Concept Recognizer`. The library of interrelated program plan templates in each approach are essentially the same as we outline in SPU, with the exception that a hierarchical structure may be imposed on the library. In Wills' approach, program components are modeled as graph grammars and are used to parse an intermediate flow graph representation of a source program. A component's make-up is constrained by it's grammar, and these components are composed in a library (of constraints). The SPU program understanding model abstracts these differing representations of components and constraints into a

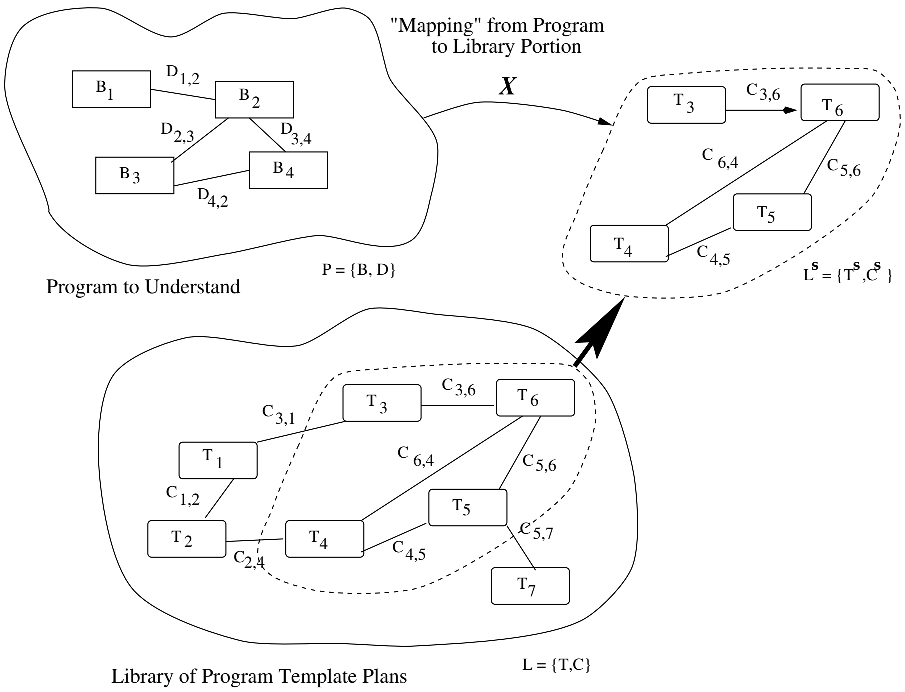

Figure 2: Simple Program Understanding.

unifying constraint-based library format. Understanding approaches uniformly assume that source programs have been pre-processed into a intermediate representation (annotated abstract syntax trees, annotated flow graphs) which makes explicit use of data-flow and control-flow information. In SPU, this information is represented as a simple program graph of related program blocks.

The SPU problem could be stated more formally as follows. Given a library of program plans $L = (T, C)$ and a source program $P = (B, D)$, does there exist at least one subgraph of the library $L^S = (T^s, C^s)$ where the templates in the subgraph $T^s \subseteq T$, and the constraints among the templates $C^s \subseteq C$, are matched to the source program by a mapping function X, defined as follows:

- X maps every program block B_i to a member of T^s; and
- X maps every program data-flow edge $D_{i,k}$ to a corresponding member $C_{u,v}$ of C^s, where $u = X(B_i)$ and $v = X(B_k)$.

We can prove the claim that SPU is NP-hard by a reduction from the **Subgraph Isomorphism** problem which is known to be NP-hard[3, p. 202]. The Subgraph Isomorphism problem may be stated as follows:

> Given a graph $G = (V_1, E_1)$ and a graph $H = (V_2, E_2)$, Does G contain a subgraph isomorphic to H, i.e., a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

The transformation to an SPU problem can be done as follows. Every vertex of V_1 in G is a program template, and every edge of E_1 in G signifies a data-flow between templates. Each vertex of V_2 in H is a program block and each edge of E_2 in H is a data-flow between blocks. A mapping between a program P and a subset of a library of related templates T exists if and only if H is a an isomorphic subgraph of G. Furthermore, this transformation can clearly be done in polynomial time.

Program Template Matching is NP-hard

Earlier we discussed that many recognition approaches attempt to recognize typical program plans or clichés, and then integrate these instances into a coherent or consistent global understanding. We have proven the simple program understanding problem is NP-hard. Now, what about the seemingly much simpler problem of finding instances of a given pattern in a program source code? In this section, we establish that even this simpler problem is NP-hard.

The **Simple Program Template Matching Problem (SMAP)** problem, is depicted in Figure 3. We are given the following:

- There exists a collection B of program blocks B_i, $i = 1, 2, \ldots, m$. These blocks can be viewed in terms of a corresponding graph $P = (B, D)$ where D is the set of edges of the graph $D_k, k = 1, 2, \ldots, n$, such that an edge $D_{i,j}$ exists between node B_i and B_j if and only if a data-flow exists between the program blocks.

- We are also given a program template plan $T = (N, C)$ where each member N_i of N participates in data-flow relationships with some other nodes in N. These relationships are specified by C, the set of edges between nodes N, such that $C_{k,l} \in C$ exists between nodes N_k and N_l if and only if a data-flow exists between the two.

Given the above structure, the SMAP problem is to determine if a mapping exists from the template nodes N to a subset of program blocks in B. The mapping is a function between relationships among template nodes and data-flows among program blocks. As an example, in Figure 3, the following correspondence gives rise to a matching instance of the program template:

$$\begin{array}{ccc} N_1 & \Longleftrightarrow & B_2 \\ N_2 & \Longleftrightarrow & B_4 \\ N_3 & \Longleftrightarrow & B_1 \\ N_4 & \Longleftrightarrow & B_3 \end{array}$$

The formal definition for the SMAP problem is similar to that for the SPU problem. We can once again prove that SMAP is NP-hard by a reduction from the NP-hard problem **Subgraph Isomorphism**, described earlier in Section 3 on page 4.

The transformation to a SMAP problem can be done as follows. Every vertex of V_1 in G is a program block, and every edge of E_1 in G signifies a data-flow between blocks. Each vertex of V_2 in H is a program template and each edge of E_2 in H is a data-flow between templates. A mapping between a template T and a subset of program blocks in a program P exists if and only if H is a an isomorphic subgraph of G.

4 Modeling Program Understanding

In this work we have outlined our approach[18, 17, 19] for representing and potentially solving the program understanding problem. This approach exploits a constraint-based methodology for recognizing program plan templates in legacy source. We have shown both the larger program understanding problem (SPU) and the template recognition problem (SMAP) to be NP-hard. Consequently, we must resort to heuristic methods for the solution of each problem. Our algorithms for each are based on the exploitation and adaptation of existing algorithms for solving *constraint satisfaction*[3] problems. In general, solution strategies are search-based, propagation-based, or hybrid.

The Modeling Process

The SPU program-understanding constraint-satisfaction problem (refered to as PU-CSP when formed as a constraint satisfaction problem), is formed in the following way. Suppose that an initial decomposition[4] of the source code is given. Each **block** of source code corresponds to a *variable* in the PU-CSP. The *variable domains* correspond to all possible explanations (mappings to the knowledge or library) of an individual block of the source code. The constraints between the variables can be specified via both the structural relationships in the source program, and subsequently, knowledge relationships in the program plan library. A *solution* is a mapping between the source code blocks and the library such that the constraints are satisfied.

We illustrate the modeling process in more detail. A Program Understanding CSP (PU-CSP) is formulated via four distinct steps shown in Figure 4. First, the legacy source is pre-processed, creating an intermediate representation which precisely captures many interrelationships among the elements of the abstract syntax tree implicit from a parsing of the source. This representation includes data-flow and control-flow between functional blocks. Second, the source code is partitioned into spatially localized, cohesive code blocks[5] which exhibit several inter-block functional relationships. Third, a skeleton CSP is formulated consisting of one variable for each identified source block, and constraints between these variables are derived from the intermediate representation level artifacts. The combination of types input and output flows into each particular block are adopted as *reflexive* constraints on the corresponding variable, effectively limiting the range of program plans that might explain that block. Finally, each CSP variable is matched (or indexed by type information) against the templates in the program plan library. Potentially matching plan templates are then composed as the domain ranges of each variable.

In a PU-CSP, the constraints among variables are of two types:

- *Structural* constraints are determined from the legacy

[3] Kumar[6] provides a good survey of CSPs.

[4] This decomposition is such as would be created from Devanbu's annotated abstract syntax tree parsing of C++ programs.

[5] These code blocks may be of varying size and complexity. The actual determination of appropriate blocking characteristics will be investigated empirically in later work. It is important to note that since the library of knowledge is arranged hierarchically it will often be the case that smaller blocks will tend to correspond to lower-level program plans and vice-versa. Consequently, the problem itself may be thought of as the need to generate a sequence of multi-layered mappings. It has been suggested by Alex Quilici in a personal communication that these mappings would best be generated bottom-up from small code fragments and plans to larger, however, this is not the only possible approach.

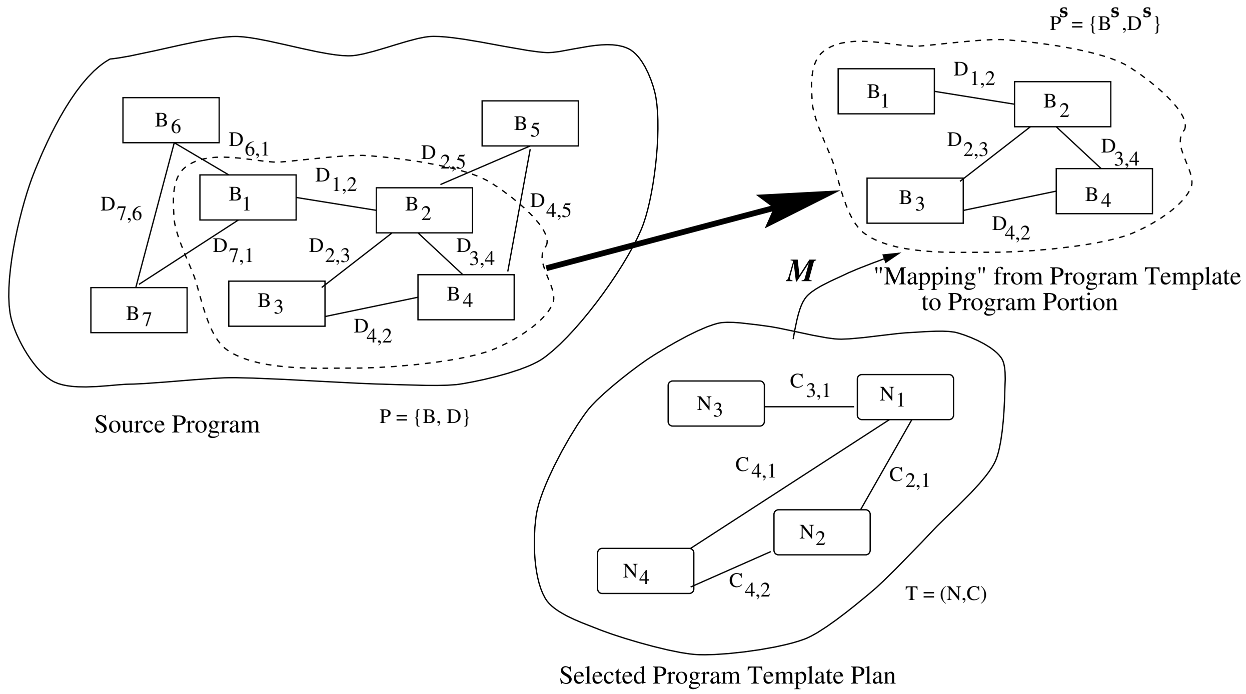

Figure 3: Program Template Matching.

code. They include such things as scope or called/calling relations, precedence relations, or shared information relations between component blocks.

- *Knowledge* constraints are independent of the legacy code. These constraints reside in the AND/OR hierarchical program plan library, restricting program plan inter-relationships. The AND connections indicate a parent-child component relationship, while the OR connections indicate specialization/generalization relations. Each of ANDs and ORs can serve to indicate important details of the parent-child relationship, such as the *role*[6] of a child as part of the higher level parent, or what details specialize a child from a parent. specializing an abstract plan in one of several ways. Assigning one program plan as an explanation of a particular PU-CSP variable thus constrains consistent assignments of other component variables. This detail effectively describes the allowable range of known program plan structure.

A solution to the PU-CSP is an assignment to each variable by one program plan component in the plan library, such that no structural constraint from the source code, or knowledge constraint from the plan library is violated.

[6] For instance, a compositional role might describe what data-flows a child provides in its function as part of a parent. In a more abstract instance, this role might be a service rather than a low-level data-flow.

The representation of program understanding as PU-CSP provides a convenient framework for the interpretation of earlier program understanding heuristics as particular constraint manipulations. For example, the Quilici-style indexing outlined earlier in which an index instance in a source code signals the need to attempt to match a particular program plan from the library can be thought of as a specific constraint ordering during CSP search. Quilici-style specialization preferences can be viewed as a heuristic for ordering the application of hierarchical knowledge constraints, essentially reducing the range of domain variables in a hierarchical CSP. Similarly, Quilici and others refer to inferences or *implications* which indicate the likely existence of plans based on the identification of other plans. Such behaviour can be interpreted as a special kind of dynamic variable-ordering heuristic in which successful instantiation of a particular variable suggests the need to attempt to instantiate a related variable next.

5 Applying Local Constraint Propagation

In general, the more constrained a particular CSP, the easier it is to solve, provided it is known in advance which parts of the problem are the most highly constrained. It is our contention that program understanding can be thought of as a well-constrained problem in many useful instances. Software is ideally well-structured and compartmented by design, and a rich system of structural constraints between functional blocks can be extracted through known meth-

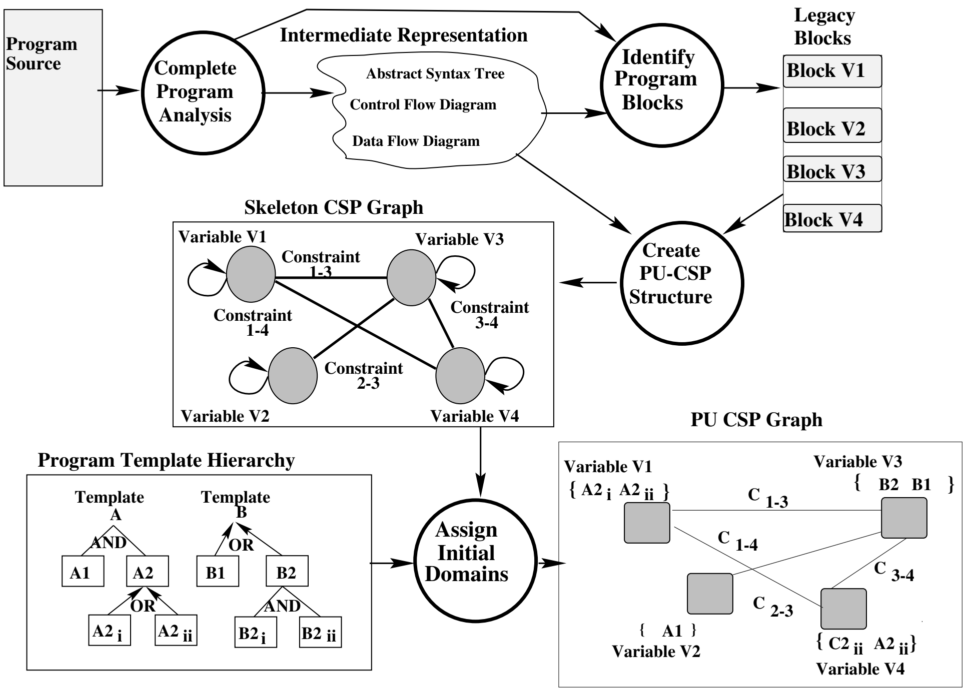

Figure 4: A PU-CSP Formulation.

ods. Program plan libraries such as commercial or shared object libraries contain a similar structure of knowledge constraints which can be annotated with design information much more readily than is the case with any particular piece of software since it has been intended for wider distribution and use.

The large number of knowledge and structural constraints in a particular problem instance combine to effectively limit the number of consistent explanations or mappings for collections of related program blocks. In particular, the application of even one such structural constraint among blocks could reduce the domain size of a program block significantly. This reduction can in turn be cascaded through adjoining block relations into successive reductions of other domains. This process is known in CSP related algorithms as local constraint propagation. An algorithm which enforces that all domains be consistent with their immediate neighbors is known as an *arc consistency*(AC) algorithm[7]. Many variations and extensions to the original AC algorithm, AC-3[7] have appeared in the literature, some of which are mentioned in [12]. These algorithms and many variations have been extensively applied and tested with a wide range of problems.

Consider a pair of variables (X, Y) and a relation $R(X, Y)$. The *arc* R is said to be *consistent*, if for every domain value of X there is at least one consistent domain value of Y. If this condition is not satisfied, a REVISE routine can be applied to the pair to remove any value of X that does not have a corresponding consistent value of Y. If, for every pair of related variables in a problem, all are consistent, the problem has been made *arc-consistent*.

5.1 An Example PU-CSP using Local Constraint Propagation

In this section we demonstrate the applicability of local constraint propagation with an example. The repeated application of local constraints to reduce variables domains admits a solution with no search in this example.

A piece of input legacy code is shown on the left of Figure 5. The code is parsed and program blocks extracted along with data-flow information as shown on the right side of the figure. This figure has been significantly simplified for the purposes of our example.

We wish to utilize the PU-CSP[8] framework to *understand* the legacy source in terms of the hierarchical program template library fragment given in Figure 6. The legacy source blocks are mapped to variables, and initial domain ranges are assigned according to block input and output types. Variable `isrt` potentially maps to several library plans based solely on input and output typing. A further (reflexive) constraint application

[7] Other algorithms enforce different degrees of consistent, from only partial arc consistency over a subset of all arcs, to consistency along paths of arcs of varying lengths.

[8] PU-CSP corresponds to the SPU model.

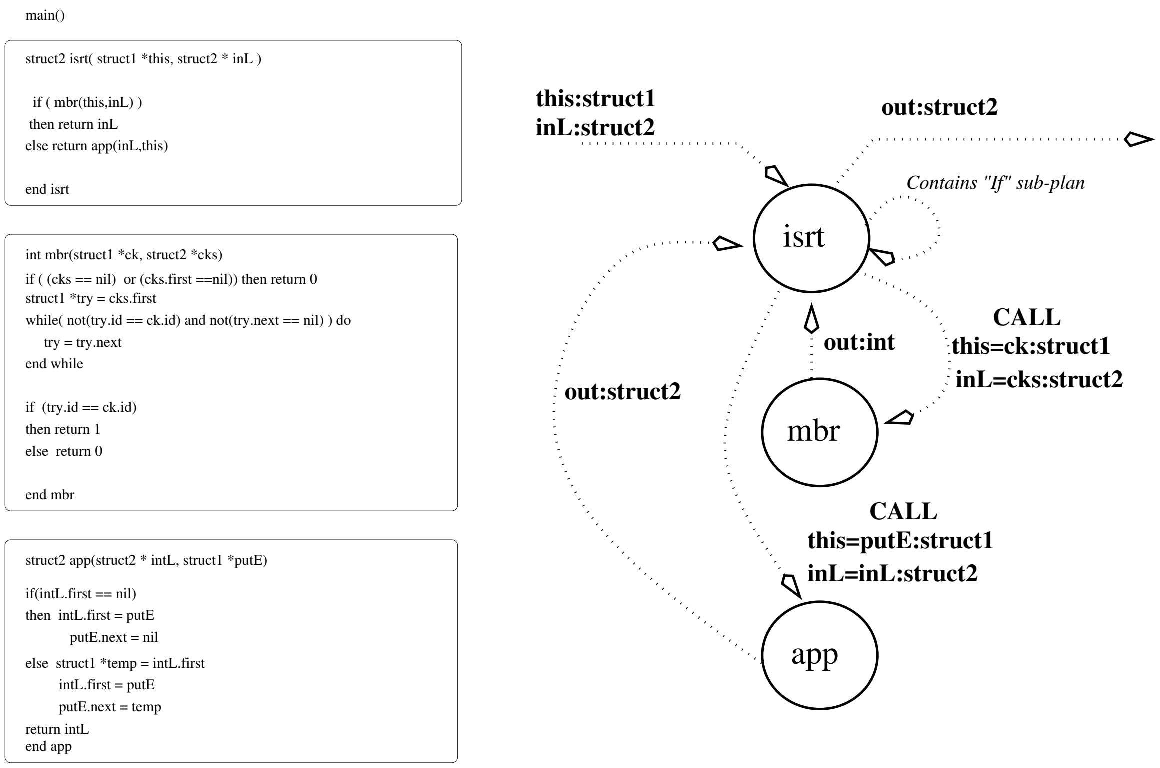

Figure 5: One Simple "Blocking" of a Legacy Fragment.

Notation	Meaning
SMAP	Simple Program Template Matching Problem
MAP-CSP	Template Matching Constraint Satisfaction Problem
SPU	Simple Program Understanding Problem
PU-CSP	Program Understanding Constraint Satisfaction Problem

Table 1: Notation Summary.

based on observation of key components (`If`)in the structure of `isrt` could significantly reduce this set. In our example, only plans $\mathbf{Insert}_{Set}$, $\mathbf{Delete}_{Set}$, and $\mathbf{Delete}_{List}$ satisfy this constraint. Similarly, `app` maps to $\mathbf{Member}_{Set}$ and $\mathbf{Member}_{List}$; `mbr` to $\mathbf{Putin}_{Set}$, $\mathbf{Putin}_{List}$, $\mathbf{Cut}_{Set}$, $\mathbf{Cut}_{List}$, $\mathbf{Insert}_{Set}$, $\mathbf{Insert}_{List}$, $\mathbf{Delete}_{Set}$, and $\mathbf{Delete}_{List}$.

The domain range of variable `isrt` may be *revised* with respect to `mbr`. In this case no values may be removed since $\mathbf{Insert}_{Set}$ is consistent with value $\mathbf{Member}_{Set}$, $\mathbf{Delete}_{Set}$ is consistent with value $\mathbf{Member}_{Set}$, and $\mathbf{Delete}_{List}$ is consistent with value $\mathbf{Member}_{List}$. Revising `app` with respect to `isrt` yields consistent mappings for values $\mathbf{Putin}_{Set}$, $\mathbf{Cut}_{Set}$, and $\mathbf{Cut}_{List}$.

After this revision, no further reductions can be made, and we are left with three combined alternate explanations that are consistent with the structure that we have outlined. These three are: (1) a set insertion plan, (2) a set deletion plan, or (3) a list deletion plan.

This ambiguity can be easily resolved if we more closely expanded the structure of `app`, showing that the structure is an insertion rather than a deletion plan on the basis of the lack of an iteration which a deletion would require. A reduction in the range of `app` would result, leaving only the value $\mathbf{Putin}_{Set}$ in the range. A revision of `isrt` with respect to `app` would now result in a singleton value $\mathbf{Insert}_{Set}$ remaining, and subsequently `mbr` could be reduced to only $\mathbf{Member}_{Set}$.

Our example problem is completed with the successful construction of a single mapping to the given program plan

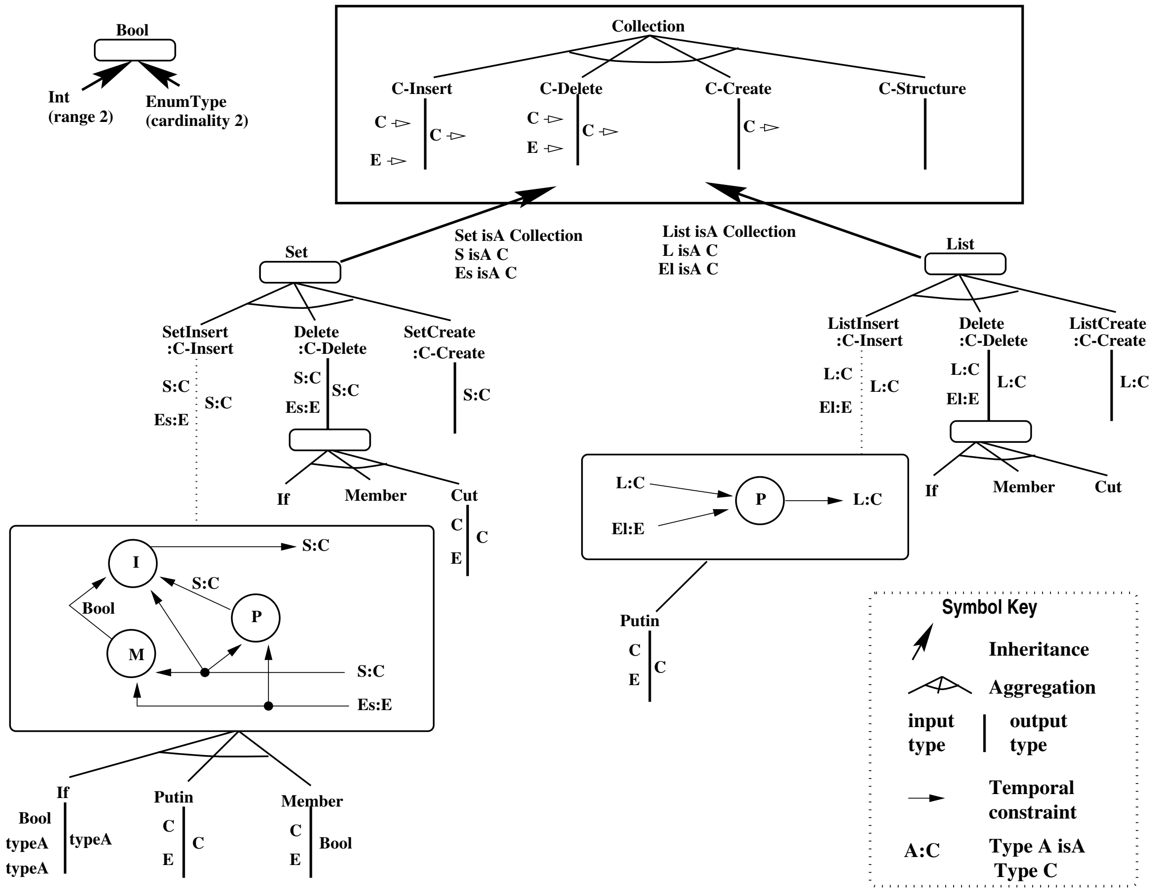

Figure 6: Library Fragment.

library. The legacy source consisting of three slices is an instance of an $\mathbf{Insert}_{Set}$ program plan with two primary subplans $\mathbf{Member}_{Set}$, and $\mathbf{Putin}_{Set}$. $\mathbf{Insert}_{Set}$ occurs in the library fragment only as a part of the **Set** abstract data type plan group, and further as part of the **Collection** abstract data type plan. No other interpretations are possible given the knowledge constraints and structural constraints of this example.

6 Implementation and Research Status

In earlier work we introduced our CSP model of program understanding, including promising empirical results in the sub-problem[9] of recognizing individual program plan templates in a range of large generated legacy examples[18, 17, 19]. As well, we have outlined the similarities and differences that we perceive between current and past program understanding methodologies and earlier plan recognition work[16]. For an overview of our terminology, pleaser refer to Table 1.

We are currently engaging in cooperation with a main telecommunications provider to investigate the applicability of this approach to extremely large source code in the telephony domain. Achieving partial automatic recognition of even a small percentage of the code would greatly benefit software maintainers.

In addition to including earlier understanding efforts in our CSP paradigm[15], our current work includes efforts to elucidate and complete implementation of the encompassing program understanding representation and algorithm (PU-CSP or SPU), accommodating the hierarchical nature of program plan libraries. These hierarchical program plan libraries give rise to interesting algorithms that must accommodate hierarchically organized domain values in constraint satisfaction. We are working towards a future dissertation in which we shall demonstrate that useful cases exist in which the inherent structure in legacy code is sufficient to allow existing constraint propagation methodologies to provide efficient solutions even in the case of large instances of "real" industrial legacy code.

[9] The SMAP implementation is refered to as MAP-CSP in other work.

7 Conclusion

In this paper we showed that the program understanding problem is NP-hard. This result provides a formal justification for researchers to look for heuristic methods for solving the problem.

We also presented a constraint-based model for program understanding, allowing much of the previous work to be cast under a unifying framework. We illustrated this model with an example and a brief overview of the associated algorithm.

Acknowledgments

We thank Alex Quilici and Toby Donaldson for their insight and comments. This research has been carried out with the support of the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre.

References

[1] Sandra Carberry. Modeling the user's plans and goals. *Computational Linguistics*, 14(3):23–37, 1988.

[2] Prem Devanbu and Laura Eaves. **Gen++** - an analyzer generator for c++ programs. Technical report, AT & T Bell Labs, New Jersey, 1994.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability : A guide to the theory of NP-Completeness*. W. H. Freeman and Company, Bell Laboratories, Murray Hill, New Jersey, 1979.

[4] Henry Kautz and James Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, Philadelphia, Pennsylvania, 1986.

[5] Wojtek Kozaczynski and Jim Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.

[6] Vipin Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, pages 32–44, Spring 1992.

[7] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[8] H. Muller, K. Wong, and S.R. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the Colloquim on Object Orientation in Databases and Software Enginering*, pages 88–98, December 1994.

[9] Alex Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[10] Alex Quilici and David Chin. DECODE: A cooperative environment for reverse-engineering legacy software. In *Proceedings of the Second Working Conference on Reverse-Engineering*, pages 156–165. IEEE Computer Society Press, July 1995.

[11] C. Rich and R.C. Waters. *The programmer's apprentice*. Addison-Wesley, Reading, Mass., 1990.

[12] P. Van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[13] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(2):113–172, February 1990.

[14] L. M. Wills. *Automated program recognition by Graph Parsing*. PhD thesis, MIT, July 1992.

[15] Steven Woods and Alex Quilici. Representing memory-based program understanding as constraint satisfaction. Technical report, University of Waterloo, Department of Computer Science, 1995.

[16] Steven Woods, Alex Quilici, and Qiang Yang. Program understanding and plan recognition: reasoning under different assumptions. Technical report, University of Waterloo, Department of Computer Science, 1995.

[17] Steven Woods and Qiang Yang. Constraint-based plan recognition in legacy code. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE)*, August 1995.

[18] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, pages 318–327, July 1995.

[19] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. Technical report, University of Waterloo, Department of Computer Science, 1995.