Program Understanding as Constraint Satisfaction: Representation and Reasoning Techniques

Steven Woods [*]
University of Hawaii at Manoa
USA

Qiang Yang [†]
Simon Fraser University
Canada

Abstract

The process of understanding a source code in a high-level programming language involves complex computation. Given a piece of legacy code and a library of program plan templates, understanding the code corresponds to building mappings from parts of the source code to particular program plans. These mappings could be used to assist an expert in **reverse engineering** legacy code, to facilitate **software reuse**, or to assist in the *translation* of the source into another programming language. In this paper we present a model of program understanding using constraint satisfaction. Within this model we intelligently compose a partial global picture of the source program code by transforming knowledge about the problem domain and the program itself into sets of constraints. We then systematically study different search algorithms and empirically evaluate their performance. One advantage of the constraint satisfaction model is its generality; many previous attempts in program understanding could now be cast under the same spectrum of heuristics, and thus be readily compared. Another advantage is the improvement in search efficiency using various heuristic techniques in constraint satisfaction.

1 Foreword

Three years have passed since the inception of the idea of applying constraint-based representation and techniques (CSP) to program understanding and design pattern recovery. The original idea was conceived by Steven Woods and Qiang Yang at the University of Waterloo in late 1994. The original aim of the work was two-fold: to provide a unifying framework for different approaches in program understanding so they can be compared, and to employ the heuristic power of well-known CSP algorithms. Since then, several different work threads have succeeded in extending the representational power and heuristic adequacy of this approach.

[*]Department of Electrical Engineering, Honolulu, Hawaii 96822. `sgwoods@spectra.eng.hawaii.edu`

[†]School of Computing Science Intelligent Software Group (ISG), Burnaby, BC V5A 1S6. `qyang@cs.sfu.ca`

This article is an extended summary of the original concepts and ideas that spun the initial research direction. Originally written in 1995, this paper's importance is underscored by the publication of numerous follow-up works. The following publications have appeared prior to the appearance of this paper:

- [46] presents an examination of the complexity of the program understanding task and a discussion of how the various problem aspects contribute to problem complexity. The paper suggests the likely success of using a constraint-based control of search as a means of mediating this complexity in practice.

- [26] provides careful modeling of the heuristic approach of a specific earlier program understanding system (DECODE) using a CSP framework. The primary results of this work were to show how the constraint-approach was capable of good scaling results in comparison to previous approaches, and to suggest how the constraint model can be adapted to exploit the heuristics developed in previous methodologies.

- [41] expands upon the CSP model's initial promising scaling results and presents new benchmarks for legacy code size and plan size for the program understanding task.

- [39], Woods' Ph.D. dissertation from the University of Waterloo, presents a comprehensive view of the CSP modeling technique and empirical results. In addition, a detailed argument is presented as to how a constraint-based model can be structured to effectively support the human-centered process of program understanding. In this work, the hierarchical nature of both plans and the understanding process is woven into the fabric of a new hierarchical algorithm for constraint satisfaction.

- [8] explores the application of constraint-based program understanding techniques to the recovery of architectural patterns. The intent of this architecture recovery was to assist in the assessment of a legacy system's overall architectural complexity in response to potential system re-engineering or re-structuring. The plan and architecture recovery paradigms are very similar in problem representation – each exploiting both program constraints (structure) and architecture pattern constraints (knowledge) to reduce the computational expense of locating patterns in large source examples.

- [28] investigates the similarities and differences between the Artificial Intelligence technologies and approaches to plan recognition in general with the more specific task of recognizing program plans and understanding legacy systems. This work demonstrated that the treatment of program understanding as plan recognition is too simplistic and that traditional AI search algorithms for plan recognition are not applicable, as is, to program understanding. In particular, it was shown that the program understanding task differs significantly from the typical general plan recognition task along several key dimensions, and the program understanding task has particular properties that make it particularly amenable to constraint satisfaction techniques. In addition, it was shown that augmenting AI plan recognition algorithms with these techniques can lead to effective solutions for the program understanding problem.

- [50], Yongjun Zhang's masters thesis at the University of Hawaii, describes an extension of the original constraint-based algorithms and an implementation of [39] to accommodate constraints extracted from structural program analysis - including data-flow and control-flow constraint annotations of abstract syntax trees.

- [27] presents new empirical evidence showing how the process of recognizing program plan templates in software can be greatly assisted through the use of strong constraints obtained through structural program analysis before searching for plan instances.

- [34] outlines how the constraint-based concept recovery process can be adapted to assist in the remediation of Year-2000 (Y2K) source code problems. The Year 2000 experiences of one of the authors is combined with the concept recovery experiences of the others to provide a new perspective on how Y2K toolsets could be extended.

- [43] is a mature, carefully restructured review of the interpretation of program understanding as a process of constraint-based concept recovery. This book brings together the entire range of material published by the authors between 1995 and 1997 in a single, comprehensive framework.

This work has been conducted with the intent of modeling program understanding algorithms and heuristics in such a way as to allow the research community to build upon previous work in an incremental fashion - ultimately arriving at a clear view of how concept recovery technology can integrate effectively and efficiently into larger reverse engineering toolsets. We believe that the move in the research community to assemble libraries of programming patterns or templates in large, shareable libraries is another necessary step in the direction of producing tools which can approach real-world application quality.

2 Introduction

Humans are particularly adept at successfully interpreting explicit representations of knowledge created by other intelligent agents. A shared understanding of the terms of reference and subject material provides a basis for this interpretation. In software engineering, experts often apply such skill to the task of *program understanding*. As shown in Figure 1, it is possible to conceptualize an expert's understanding of a given source program as a successful construction of a *mapping* between the expert's store of relevant knowledge and the structures and components inherent in the source code. The expert or agent can use this mapping to infer the source program's high-level goals. This mapping essentially raises the level of abstraction of the understanding of the source from the level of actual code to the more abstract level of the existing representation (or language of expression) of the domain knowledge. This abstract understanding may be exploited as part of the process of: (1) translating the program into the source code of another programming language, (2) recognizing errors in the legacy code and assisting in debugging the code at the more abstract level, and (3) replacing understood code portions with generic application code or calls to other code libraries. We know that in many real-world circumstances, a reduction in the size of an existing source code library by only a small percentage can result in a substantial reduction of the maintenance cost. Consequently, the creation of even a partial mapping between existing domain knowledge and a particular legacy source can be a valuable tool for maintenance or re-writing engineers.

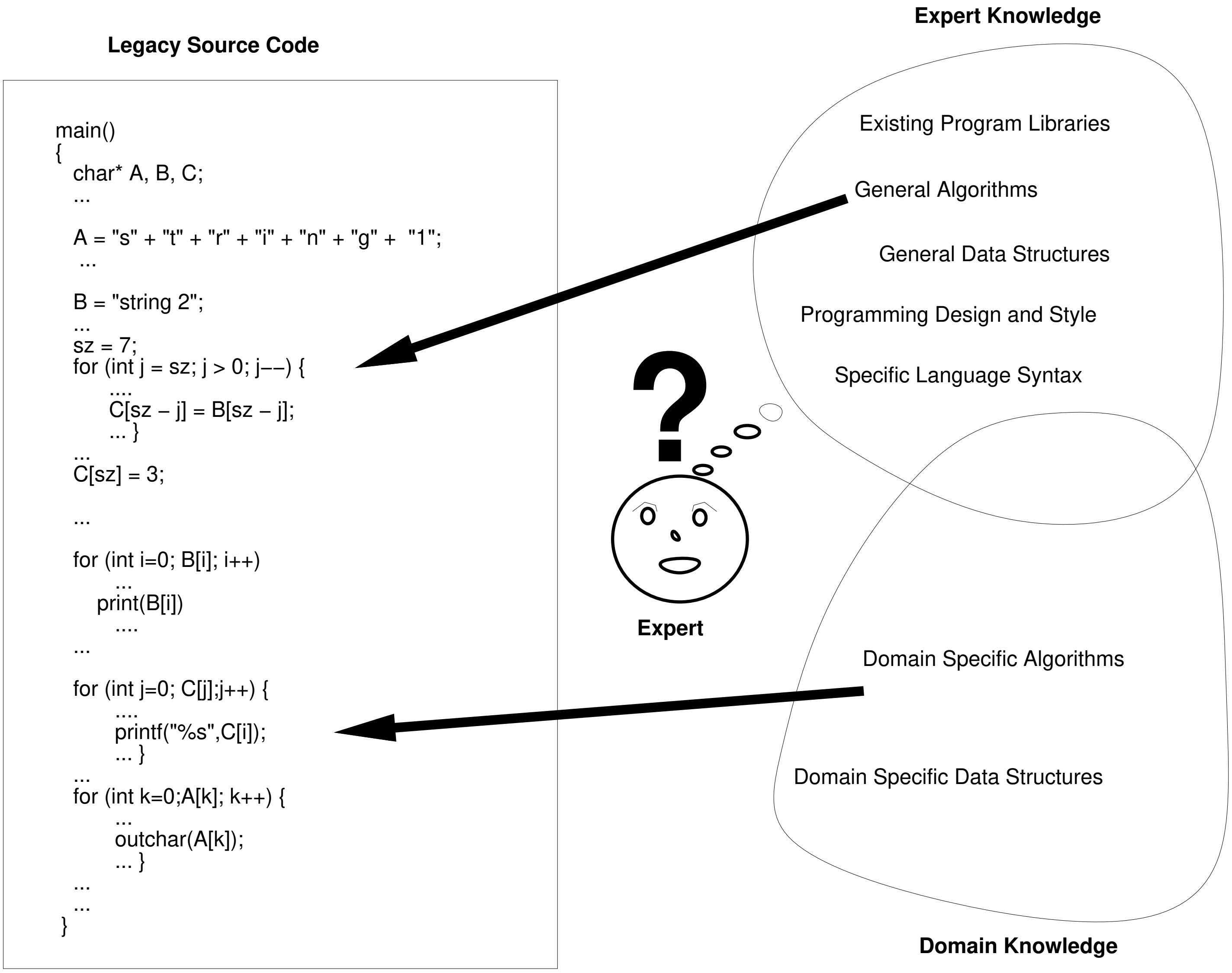

Figure 1: Conceptualizing source with expert knowledge.

In Artificial Intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition [7, 1, 2, 33]. In much of this work, existing human knowledge in a particular domain is represented as hierarchies of plans that describe relevant actions and goals. Given such a plan hierarchy, and example of which is shown in Figure 1 on page 4, plus an observation of another agent's plan, a plan-recognizer would typically construct a mapping from input plan fragments to the leaf nodes of the knowledge-base and infer upwards toward a goal. To disambiguate among alternative goals, the mapping processes may employ knowledge about the temporal relations between parts of the plan. These plan recognition programs have been applied mostly to *toy domains* (such as the cooking domain), involving small knowledge bases and a small search space. The plan recognition and program understanding approaches have been compared in some depth elsewhere[42].

Recently, researchers have adopted a more direct approach to program understanding. In this direction, 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. Notable examples are Quilici[22], Kozaczynski and Ning[11], Rich and Waters[30] and Wills[36, 37]. To some extent, all are aimed at improving the effectiveness of the mapping process through heuristic knowledge. The basis for such heuristic approaches has been the assumed intractability of the complete understanding problem in general. In [47], not only is program understanding is shown to be NP-hard, but also the intuitively easier problem of locating partial local understandings.

In Figure 2 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, in this case as an instance of the plan template **copy-character** specified in the library.

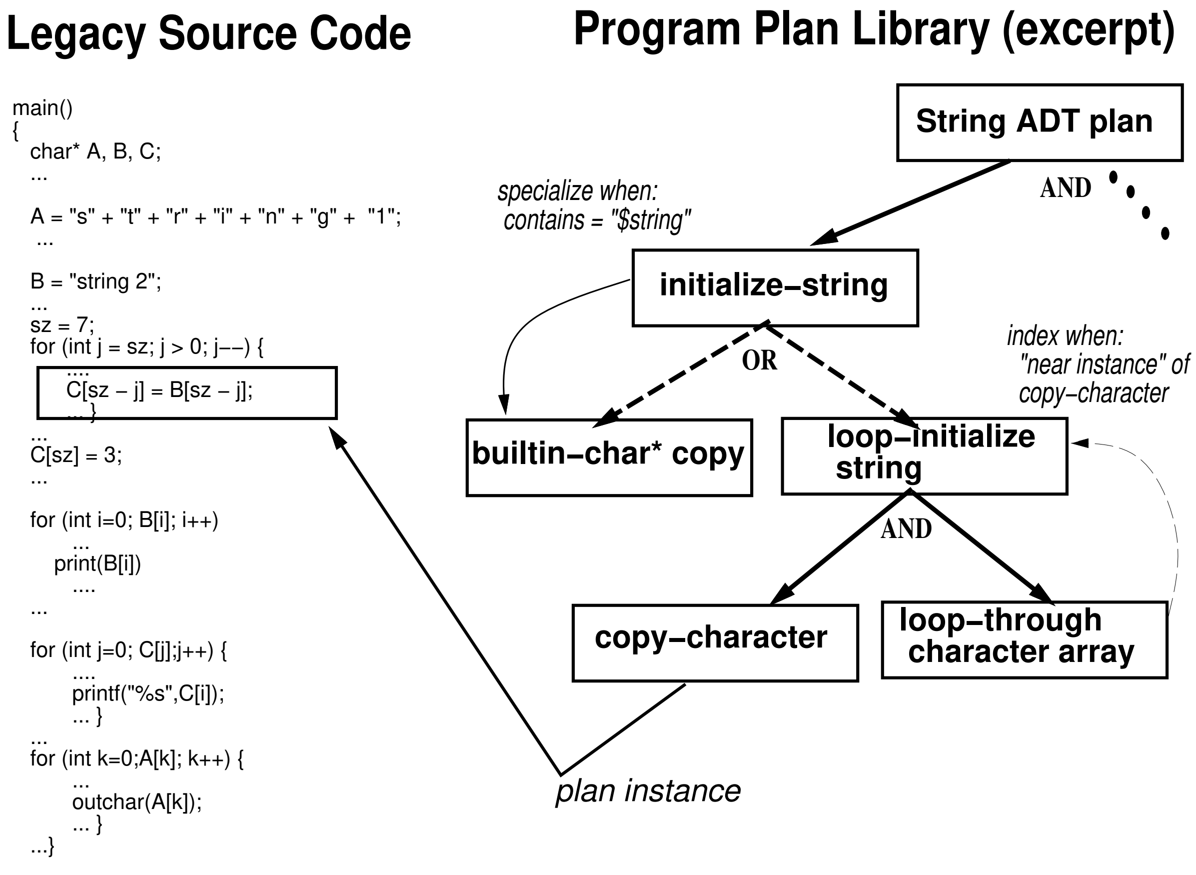

Figure 2: Conceptualizing source with a plan library.

Much of the previous program understanding work has failed to demonstrate heuristic adequacy in even partially generating "understanding" of large problems. Specifically, many recognition algorithms presented may be viewed as collections of heuristic tricks. This construction makes it difficult for one to perform a systematic analysis of different search methods within a particular approach, or to understand how the addition or deletion of certain types of domain-specific knowledge may affect performance. We are unaware of concrete examples or experiments which might suggest that these approaches might scale up for specific uses in large sources. One exception might be Wills[37] who presents empirical results which seem promising in identifying partial mappings of reasonably sized legacy sources to a library of program plans.

The work presented in this paper is part of the initial phase of work focused on demonstrating that an effective approach to partial program understanding is possible with large legacy code examples. Specifically, we intend to clearly categorize the circumstance in which this use is possible, and the preconditions which must first be met in terms of representation and application of domain knowledge. We present a generalized representation of program understanding as a *Constraint Satisfaction Problem (CSP)*[13]. For a given legacy source code, the program components (explained later) are variables in the CSP. The domain values are the known program plans that may *explain* each component. The CSP constraints are either *knowledge constraints* which describe how program plans may fit together to form larger plans, or *structural constraints* which describe how program components are structurally related. We refer to the program understanding CSP as PU-CSP.

In addition, we present and empirically evaluate a mapping algorithm (as part of the PU-CSP), also formulated as a CSP, which provides the ability to locate all instances of a specific general programming plan template, and to map the plan's structure to actual source program components. We refer to this mapping CSP as MAP-CSP. Some earlier works also attempt to define and recognize abstract concepts as part of program understanding[11, 37]. For a given program plan template (explained later), the different parts of the template are the variables in the MAP-CSP. The various syntactically known pieces of the source code correspond to domain values for each variable. The constraints among the different parts of the program plan are constraints in the MAP-CSP.

There are at least two advantages in our constraint-based approach. The first is its **generality**; most of the previous recognition methods and heuristics can now be unified under the constraint-based view. Another advantage is an increased ability to address **heuristic adequacy**, or **scalability**; by casting program understanding as a CSP, the previously known constraint propagation and search algorithms could be easily adapted. We may now perform a systematic study of different search heuristics, including both top-down and bottom-up as well as many other hybrids, in order to discover their applicability to a particular source code.

The rest of this paper is organized as follows. Section 3 outlines the program understanding problem including an illustrative example and reference to previous approaches. Section 4 provides an introduction to our representational model, constraint satisfaction and delineates the two primary sub problems in program understanding. Section 5 describes how the larger sub problem of explaining source block interrelationships (PU-CSP) is modeled using constraints. Section 6 details how the sub problem of identifying individual source code template instances (MAP-CSP) may be modeled. Section 7 presents empirical results from experiments with MAP-CSP. Section 8 presents our conclusions and indicates our current research directions.

3 The Program Understanding Problem

3.1 An Illustrative Example

Consider the C program outlined on the left hand side of Figure 3. This example program contains declarations, initializations and an embedded print loop for *each* of three strings. As an illustration, strings are treated as a primitive data type by the programmer, with no shared functionality for printing.

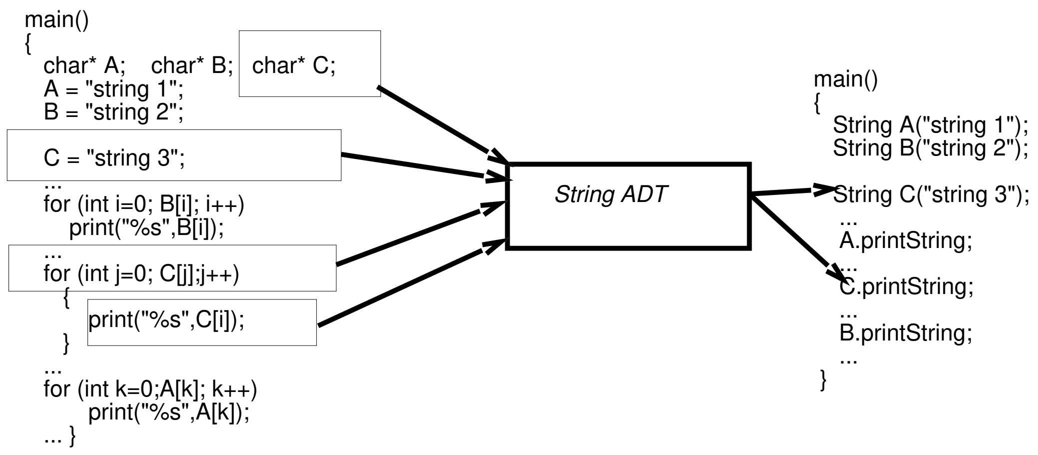

Figure 3: C legacy code mapped as String ADT instance to C++ code.

To understand this program, one might use as a basis a library of program plans as shown in Figure 4 which represents previously compiled knowledge about program composition within a particular domain. Table 1 shows a program plan for the Abstract Data Type (ADT) or class **String** which is part of this library of plans. Once a mapping is constructed between the source and compiled knowledge, one could translate the redundant source code to one with a single inclusion of the ADT, as shown in the C++ code on the right hand side of Figure 3.

Given the legacy source code on the left side of Figure 3, we would like to *understand* or *explain* some portions of the source program within the known context of the program plans such as represented by the **String** ADT. Successful identification could result in the replacement of much redundant source code with a single inclusion of the ADT. The C++ code shown at the right of Figure 3 is obtained with replacement of C source with references to **String** ADT functionality. This understanding process might be executed in two steps. First, one identifies all instances of a particular abstract program plan in a source code. We refer to this problem as the *MAP-CSP* problem. Second, one relates some set of identified plan blocks (or program slices) to conform to the hierarchical structure in a given program-plan knowledge base. The latter we refer to as the *PU-CSP* problem.

```
Class String  {
 char localStr [MAXSIZE];

 String( char* inStr )
 {
   for (int j=0; inStr[j]; j++)
        localStr[j] = inStr[j]; }

 printString()
 {
  for (int j=0; localStr[j]; j++)
      printf("%s",localStr[j]); }   }
```

Table 1: Example abstract data type.

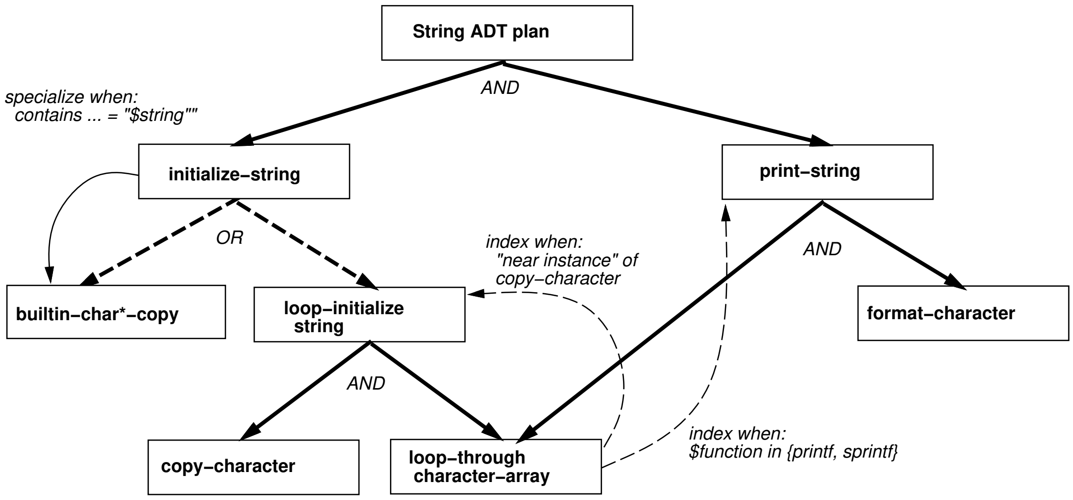

Figure 4: **String** ADT within a hierarchical program plan library.

We identify two important benefits of locating mappings between a programming plan library and an existing source or legacy code. First, the resulting replacement of legacy code with ADT instances can result in substantial reduction in code. This size savings can reduce the amount of effort required for subsequent code understanding or maintenance by programmers. Second, the mapping between source and library plan can be used as a building block in attempting to understand and translate the legacy code. The intent of this work is twofold. We describe how various types of individual mappings can be identified efficiently, and we outline how this mapping process may be integrated into the larger task of program understanding.

3.2 Quilici's Memory-based Method (Decode)

Quilici's method is representative of other earlier work in this area, including work by Kozaczynski and Ning[11]. This approach [22, 24, 23, 25] is based on a construction of an explicit library of programming plan templates, complete with an indexing ability, which can quickly associate a particular instance of recognized source code with program plan templates in the knowledge base. Furthermore, a combination of top-down and bottom-up search strategies is utilized to implement the matching process. With this system Quilici demonstrated how simple C programs could be translated to C++ programs.

Program plans (such as embedded in ADTs) are organized hierarchically in a library as shown in Figure 4. Legacy source code in the form of an abstract syntax tree is mapped to the plan library through the use of indices, which are pointers from the source code to parts of the plan library. Index tests indicate when to *specialize* or to *infer* the existence of other plans according to a set of conditions. As an example of specialization, consider Figure 4 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 4, 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.

Given a source code and a program plan, Quilici describes an approach to understanding the legacy source based on a search in the plan library. Search behaves *bottom-up* when existing index tests indicate possible higher-level explanation plans for a particular lower-level component in the library. Quilici observes that people only make bottom-up inferences in particular "well-known" circumstances, and consequently limits the number of upward explanations by inferring only those specified by explicit indexes. On the other hand, search behaves *top-down* when low-level components are indexed and subsequently matched based on some hypothesized high-level plans. Quilici's algorithm attempts to specialize any matched plan as much as possible according to predefined specialization tests, and directs search for low-level plans based on high-level hypothesized plans. This approach marks one of the first cognitively motivated attempts to program understanding using a hierarchical library of program plans. There are, however, a number of shortcomings. First, the lack of a general mathematical model of the indexing and search process makes it unclear as to how one should coordinate the top-down and bottom-up search. Second, Quilici's algorithm depends on a number of heuristics, such as specializing a plan as much as possible. It is not clear how these heuristics integrate or how they scale-up when the problem size increases. Finally, Quilici makes a substantial effort in capturing actual programmer's methodologies as heuristic enhancements to search control, but presents no empirical results.

While studying this work, it occurred to us that the program understanding problem could be broken down into a number of choice points. Examples of these choices include: (1) choosing among candidate unexplained components, (2) choosing among multiple initial plan assignments for a component, (3) choosing among several plans whose existence is implied top-down, and (4) choosing a particular index or specialization test from a candidate set. The existence and interactions of these decisions are buried in Quilici's presentation, but are very important in addressing the efficiency of the search problem. In the next section, we explore how to represent and exploit these choice points using a simple and elegant mathematical model known as *constraint satisfaction*. A more detailed treatment of Quilici's approach in terms of constraint satisfaction (known as Memory-CSP) is provided in [26, 40] and elaborated further in [39].

3.3 Wills' Graph Parsing Method

Wills[30, 36, 37] outlined an approach to recognition in which stereotypical program or data structures known as *clichés* are represented as a type of graph grammar. A source program is translated into an intermediate representation as a flow graph. These flow graphs are parsed to identify all possible derivations of the flow graph based on the known *clichés*. These derivations each represent a possible *partial* interpretation of the source program or mapping to the library of clichés. Wills notes that although the parsing problem is NP-complete in general, experience suggests that attribute constraint checking significantly prunes the search space. Wills evaluates the effectiveness of such an approach empirically for two medium-size source code examples.

Wills' work differs from our approach in at least 3 important ways: (1) cliché and program representation, (2) library knowledge representation and exploitation during search, and (3) method of integrating cliché instances in the larger understanding problem.

3.4 Other Related Work

Kozaczynski and Ning[11] describe a method of automatically recognizing abstract concepts in source code given a library of concepts and rules for how to recognize the higher-level concepts in lower-level language constructs, essentially controlling the concept search in a top-down fashion. Muller and others[18, 17, 19] are involved in the construction of Rigi, a system for analyzing software systems which includes visual representations of data and control flow structures in code allowing the identification of subsystems and hierarchies of structure in code. Kontogiannis[10, 9] has built an abstract pattern matching tool using the REFINE[1] code analyzer. This approach attempts to identify probable matches using Markov models.

Rich and Waters[29, 30] 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[30][pp.171-188] describe a cliché recognizer Recognize based in KBEmacs. Rugaber, Stirewalt, Wills and others are part of an effort in reverse engineering being conducted at the Georgia Institute of Technology. Recent work[31] describes one major research area in program understanding known as interleaving in which program plans intertwine.

4 An Introduction to Constraint Satisfaction

Constraint satisfaction problems (CSPs)[13, 12, 32] provide a simple and yet powerful framework for solving a large variety of AI problems. The technique has been successfully applied to machine vision, belief maintenance, scheduling, and planning, as well as many design tasks. For a successful application of this technique to knowledge-based planning, see [49].

A constraint satisfaction problem can be formulated abstractly as three components:

[1] REFINE is a trademark of Reasoning Systems

1. a set of **variables**, $X_i, i = 1, 2 \ldots n$,
2. for each variable X_i a set of values $\{v_{i1}, v_{i2}, \ldots v_{ik}\}$. Each set is called a **domain** for the corresponding variable, denoted as $Dom(X_i)$,
3. a collection of **constraints** that defines the permissible subsets of values to variables.

The goal of a CSP is to find one (or all) assignment of values to the variables such that no constraints are violated. Each assignment, $\{x_i = v_{ij_i},\ i = 1, 2, \ldots, n\}$, is called a **solution** to the CSP.

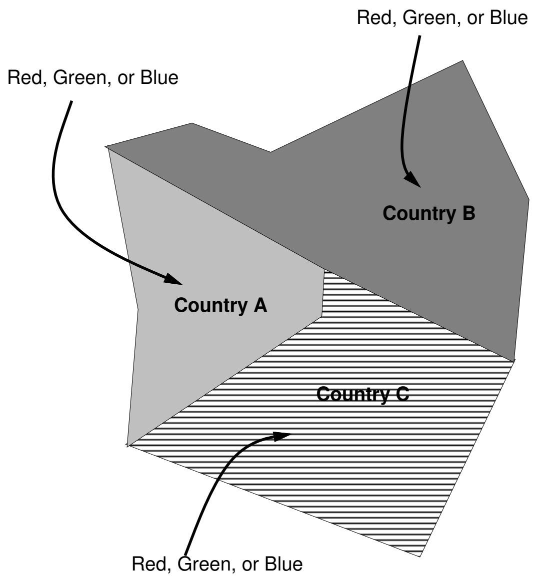

Figure 5: A Map Coloring Problem.

As an example of a CSP, consider a map-coloring problem, where the variables are regions $R_i, i = 1, 2, \ldots, n$ that are to be colored (see Figure 5). In a final solution every region must be assigned a color such that no two adjacent regions share the same color. A domain for a variable is the set of alternative colors that a region can be painted with. For example, a domain for A might be $\{\text{Green}, \text{Red}, \text{Blue}\}$. A constraint exists between every pair of adjacent variables, which states that the pair cannot be assigned the same color. Between adjacent regions A and B, for example, there is a constraint $A \neq B$. A solution to the problem is a set of colors, one for each region, that satisfies the constraints.

Let `Vars` $= \{X,\ Y,\ \ldots Z\}$ be a set of variables. A constraint on `Vars` is essentially a *relation* on the domains of the variables in `Vars`. If a constraint relate only two variables then it is called a **binary constraint**. A CSP is binary if all constraints are binary. For any two variables X and Y, we say $X = u$ and $Y = v$ is **consistent** if all binary constraints between X and Y are satisfied by this assignment.

A variety of techniques have been developed for solving CSPs. They can be classified as local *consistency-based methods*, global *backtrack-based methods* or *local-search methods*. Local-search methods [15] is a kind of greedy algorithm which is gaining popularity. We do

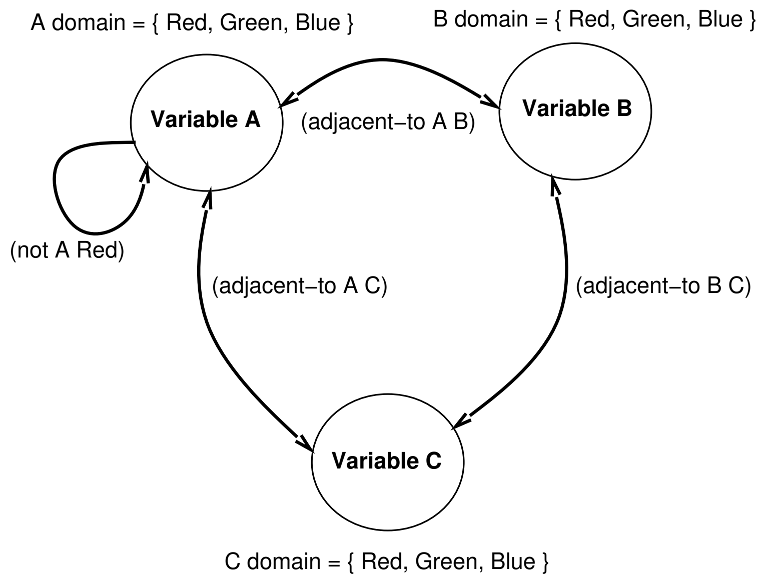

Figure 6: Map-Coloring CSP.

not review this method here, but we do intend for our CSP modeling to be general enough to include local-search as a reasoning method.

4.1 Local Consistency Methods

Local consistency methods follow the theme of *preprocessing*. That is, before a more costly method is used, a consistency-based method could be applied to simplify a CSP and remove any obviously incompatible values. Often these methods yield tremendous headway toward eventually solving the problem.

Let X and Y be two variables. If a domain value A of X is **inconsistent** with all values of Y, then A cannot be part of a final solution to the CSP. This is because in any final solution S, any assignment to X must satisfy all constraints in the CSP. Since $X = A$ violates at least one constraint in all possible solutions, A can be removed from the domain of X without affecting any solution.

If for a pair of variables (X, Y), for every value of X there is a corresponding **consistent** value of Y, then we say (X, Y) is arc-consistent. By the above argument, enforcing arc-consistency by removing values from variable domains does not affect the final solution. The process of making every pair of variables arc-consistent is called *arc-consistency*.

4.2 Backtrack-based Algorithms

Arc-consistency algorithms only work on pairs of variables, and as such can only handle binary constraints and cannot always guarantee a final solution to a CSP. A more thorough method for solving a CSP is backtracking, where a depth-first search is performed on a search tree formed by the variables in the CSP. A thorough examination of these techniques can be found in [20] and [12]. During a backtracking search, each variable instantiation is interpreted as extending the current understanding of a legacy program one step further.

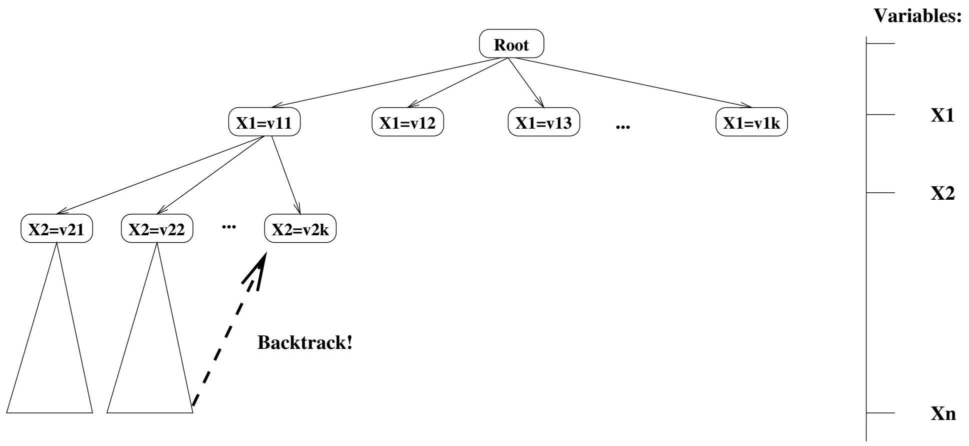

Figure 7: A search tree for a backtrack-based algorithm.

A backtracking algorithm instantiates the variables one at a time in a depth-first manner. It backtracks when the constraints accumulated so far signal inconsistency. In Figure 7 we show this process. First, variables are ordered in a certain sequence. Different orders of variables might entail different search efficiency, and heuristics for good ordering of variables are called **variable-ordering heuristics**. Similarly, for each variable, the values are tried out one at a time, and the heuristics for a good ordering of values are called **value-ordering heuristics**.

Using the CSP representation, we can also consider a more systematic study of different search algorithms. Figure 2 provides a general backtracking algorithm for solving a CSP. In this algorithm, we have a number of hooks where we could place different search heuristics. They correspond to heuristics for ordering variables and constraints, as well as heuristics for deciding the amount of constraint propagation.

There are several choice points which both individually and in combination affect the resulting search performance. These choice points are explained as follows:

1. *Initialization* and *Initial Constraint Propagation* is the determination of variables and domain values before the search starts. This can be viewed as a special type of localized constraint propagation algorithm, but one that is directed according to pre-defined domain knowledge. The determination of the set V and of $Dom(X)$ controls how much work is done in advance. This reduction could also be performed as an in-search propagation at Step 6 of the Generic CSP algorithm.

2. *Constraint Propagation* is the reduction of domains locally or globally within the CSP problem graph. Existing algorithms include AC-3[13], AC-4[16], AC-5[35], and other variations[4, 3].

3. *Variable Selection* is the determination of which component variable should be chosen next for instantiation during search. The decision may be based on domain independent measures, such as the size of a variable's domain; on information specific to the instance

Generic CSP Search
V: variables in a CSP, $Dom(X)$: the domain values of X.

1. **[Initialization] for** each variable $X_i \in V$, find the set of domain values for X_i;
2. **[Initial Constraint Propagation]** Reduce $Dom(X)$ by constraint propagation.
   Solution = NULL
3. **[Variable Selection]** Select and remove a variable X from V
4. **[Value Selection]** Select and remove a value of X from $Dom(X)$.
   The value must be consistent with all assignments in Solution.
5. **[In-search Propagation]** Apply a subset of constraints to V.
6. **[Backtrack Point Selection]** Backtrack if any $Dom(X)$ in V becomes empty.
7. **[Solution Evaluation]** If V is empty, exit with Solution (if all-solution, continue);
else, goto Step 4.

Table 2: Generic CSP Search Algorithm.

and domain plan library, such as frequency of occurrence of particular plan templates in the variable domain set, or on some combination of these types of information.

4. *Domain Value Selection* is the determination of a particular plan explanation, taken from the plan library, to assign to the component variable. Typically this selection should be made to most effectively limit the remaining variable ranges, that is, to be the most context limiting. In terms of our plan library this means a plan that is as *specific* as possible.

5. *In-search Propagation* is the reduction (as for Step 2) of the remaining uninstantiated variable domains according to some constraint propagation algorithm. Problem characteristics such as variable domains that exceed some average or absolute bounds are potential signals that constraint propagation may be useful before continuing search. In [20] the advantages of exploiting various algorithms for achieving a limited degree of partial consistency amongst variable sets are examined.

6. *BackTrack point selection* is the determination, after it has become evident that no possible solution exists along a particular variable-instantiation path, of which instantiation to retract. Intelligent backtracking approaches such as BackJumping and BackMarking[2] attempt to determine the origin of the conflict that caused the failure, and to BackTrack as far up the search tree as possible to avoid a repeated failure of the same condition.

7. *Solution Evaluation* determines whether or not a particular solution is satisfactory. In a cooperative interactive approach to program understanding, it is at this point that an expert might interact and evaluate a particular partial solution for adequacy. Similarly,

[2]These and other intelligent backtracking algorithms are described in detail by Nadel in[20].

if there exists particular measures of adequacy or *soft, preferential* constraints that may have been relaxed during search, such measures may be applied here. A complete strategy identifies *all* possible solutions, however, it is possible to identify only some set number or even 1 solution.

There are in addition several other ways to improve the search efficiency. One method is to employ the particular hierarchical structure of the plan library, and using a *hierarchical constraint satisfaction algorithm*[14]. In this approach, the plan library represents plans at varying levels of abstraction. A set of low-level program components which have been mapped to the program library may be grouped according to their functional relationships and form a higher-level component. This component (or variable) may now be explained by a more abstract plan (or domain value) according to both the structural constraints imposed in source structure and the knowledge constraints present in the program plan library. We plan to pursue this type of constraint application more completely in future work.

Another way of improving the search efficiency is to use the MAP-CSP version of the algorithm as a subroutine of the PU-CSP algorithm. This could be done at the beginning of the generic search algorithm, in Step 1. By performing a MAP-CSP for several *key* plan templates in the library up front, it may be possible to reduce the total number of domain values for each variable through constraint applications. In terms of search, this could result in an substantial amount of savings, and consequently, improved performance.

In the generic search algorithm, a set of choice points are presented in the new context of CSP solving. In the next section of this paper we discuss and evaluate several selection variations for recognition of one particular template in sets of generated source code examples. We examine variations that include applying AC-3 as Step 1 combined with BackTracking and also another more intelligent search algorithm known as Forward Checking[5], which performs a limited amount of in-search propagation at Step 6. In addition, the intelligent search algorithm dynamically rearranges the order of variables during search according to the size of the variable domains, selecting the shortest first.

The order in which constraints are applied can also dramatically affect search. Constraint ordering or selection would occur at Step 6. In particular, it is advantageous to apply constraints that are inexpensive computationally and that (potentially) prune a large number of domain values. In a particular domain it may be possible to determine or estimate such relative benefits either from past empirical results or through analysis of the domain structure itself. For instance, the property that program template features tend to be found *spatially* near each other can be exploited through heuristics that limit the range of search for related components. The effectiveness of such abstraction heuristics has been reported elsewhere[6, 38].

4.3 Program Understanding as CSP

We view the entire program understanding problem as a constraint satisfaction problem. In this model, a long program code is first divided into blocks, where each block is a set of closely related source code. The program understanding problem is to identify the top-level function of each of these program blocks, so that not only the inter-relationships between the blocks are explained, but also the constraints specified by a program library on the program plans are respected. A key problem, then, is to assign one plan component to each block, subject to a set of constraints. This problem we call the **program-understanding CSP**, or PU-CSP.

The number of program plan components that one could assign to each block could be enormous. To be practical, it is crucial to first reduce the number of explanations for each block as much as possible. This process could be helped by a related constraint satisfaction problem, one that we will explain in detail in Section 6: the problem of finding all instances of a given program plan or pattern in the entire source code. This problem we call the MAP-CSP problem.

Below, we explain both problems in detail.

5 Program Understanding CSP: PU-CSP

PU-CSP is formed in the following way. Suppose that an initial decomposition or slicing 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 of an individual source code block. As an example, consider the legacy code program statements of Figure 3 as the blocks. We take each block as a PU-CSP variable which ranges over all possible program plans of corresponding statement type, such as "declaration", "assignment", "print", etc, in the plan library of Figure 4.

5.1 The Modeling Process

A Program Understanding CSP (PU-CSP) is formulated via four distinct steps shown in Figure 8. First, the legacy source is pre-processed creating a set of artifacts that describe some precise interrelationships in the source regarding data flow relationships between functional blocks, control flow among the functional blocks, and the creation of an abstract syntax tree in an intermediate abstract language via parsing of the source. Second, the source code is partitioned according to existing program slicing methodologies into spatially localized blocks of code which are known to exhibit functional relationships among one another, and cohesive properties within one's boundaries. 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. Each variable 'typed' via the addition of reflexive constraints on the variable which describe properties of the block such as *kinds* of input or output. Finally, each CSP variable is compared against the templates in the program plan library, with any templates which potentially match a variable with regards to input and output typing are composed as the domains of that variable.

Figure 9 shows an example formulated PU-CSP in which the domains of each variable are shown as instances identified in the program template hierarchy. During discussion of the PU-CSP we will discuss two distinct types of constraints: *structural constraints* depicted in Figure 9 as the inter-variable constraints, which are exactly those constraints derived from the intermediate source representation and which describe how program components are structurally related, and *knowledge constraints*, depicted in the figure as the compositional and specialization constraints in the program template hierarchy, which describe how

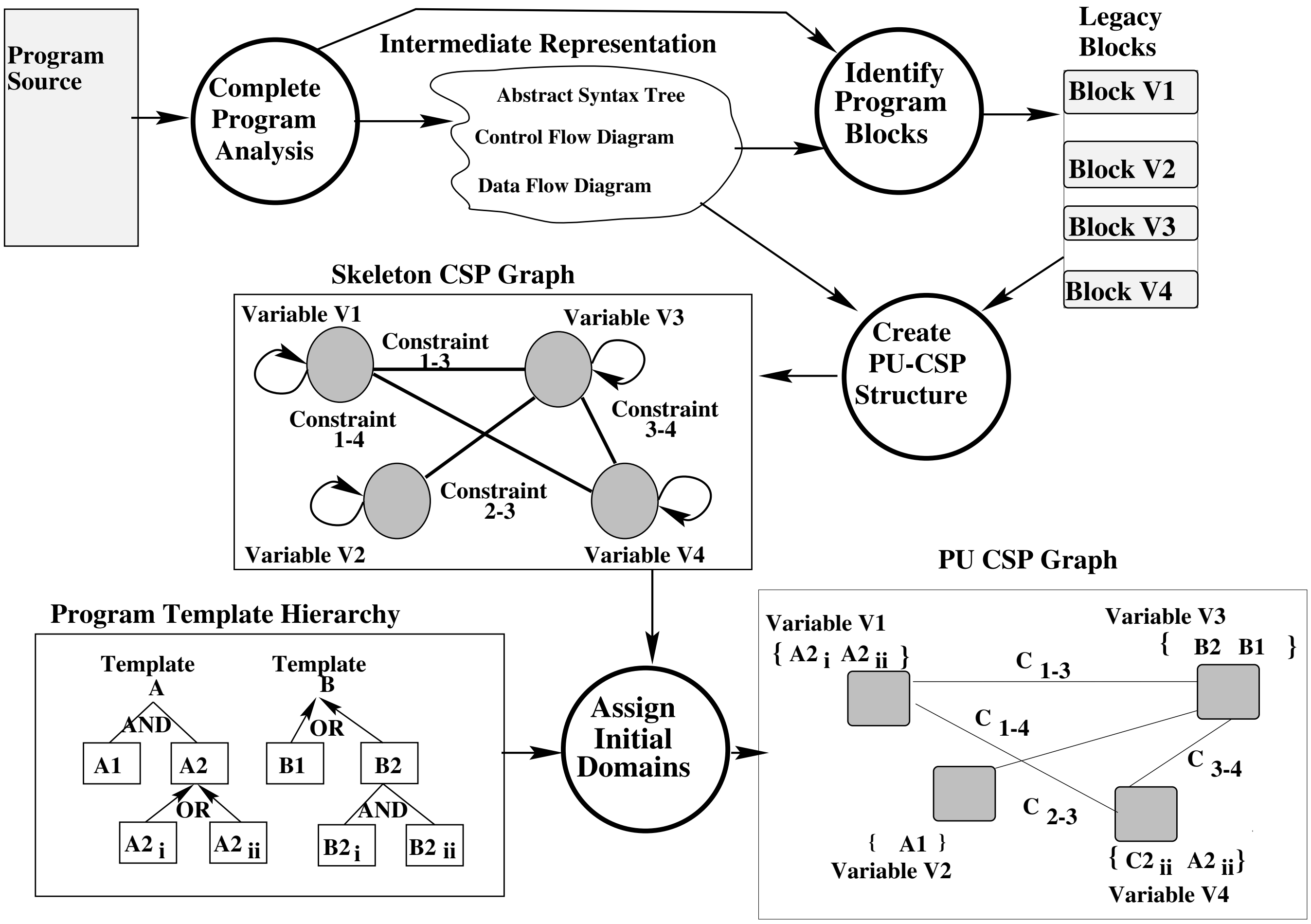

Figure 8: PUCSP Formulation; CSP Graph exploded in Figure 9.

program plans or templates may fit together to form larger (more abstract) plans in this domain.

The program template hierarchy is composed of hierarchically related plan templates (for a formalization of hierarchical planning knowledge base, see [48]). A template plan may be broken down into several sub-plans, in which case this is recorded as an **And** relationship between the sub-plans and the parent plan. Further, any required structure between the sub-plans such as necessary ordering, data flows between the sub-plans or control-flow between the sub-plans is recorded with the **And** relationship. Similarly, a template plan may be a specialization of another plan (or one of many such specializations), and in this case the constraints that constitute the specialization such as restriction of variable type or a particular restriction of data or control flow is recorded with the **Or** relationship. Figure 10 shows a simple **And** example in which **Template A** is composed of two subplans A_1 and A_2 where A_1 provides the data flow r which A_2 requires, and a simple **Or** example in which **Template A** may be specialized by either of the plans B_1, which also exports n in addition to the primary exports of B or B_2, which exports p.

5.2 More on Constraints

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

- *Structural* constraints are determined from the legacy code. They include such things as scope or called/calling relations, precedence relations, or shared information relations between component blocks. For instance, in the legacy source in Figure 3, the **print** statements appear within the scope of **for** statements, **declarations** precede their initial **assignment**, and print statements act upon array positions indexed by corresponding **for** statement indexes.
- *Knowledge* constraints are independent of the legacy code. They are program plans restricted in their relationship by the AND/OR structure given in the plan library. AND constraints are for composing program plans into higher level plans, and OR's are for 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.
  As an example of a knowledge constraint mandated from the library structure, if a variable corresponding to program component **A = "string 1"** in Figure 3 were instantiated to program plan **builtin-char*-copy** as shown in Figure 4, then it is consistent to assign the last **for-loop** variable an explanation of **print-string**, where the strings are the same.

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.

Representing program understanding as PU-CSP provides a convenient framework for interpreting Quilici's index tests as constraint applications as part of search strategies typically used for solving CSPs. Specialization tests are specific instances of knowledge constraints that may be used to systematically reduce the range of domain variables in a hierarchical

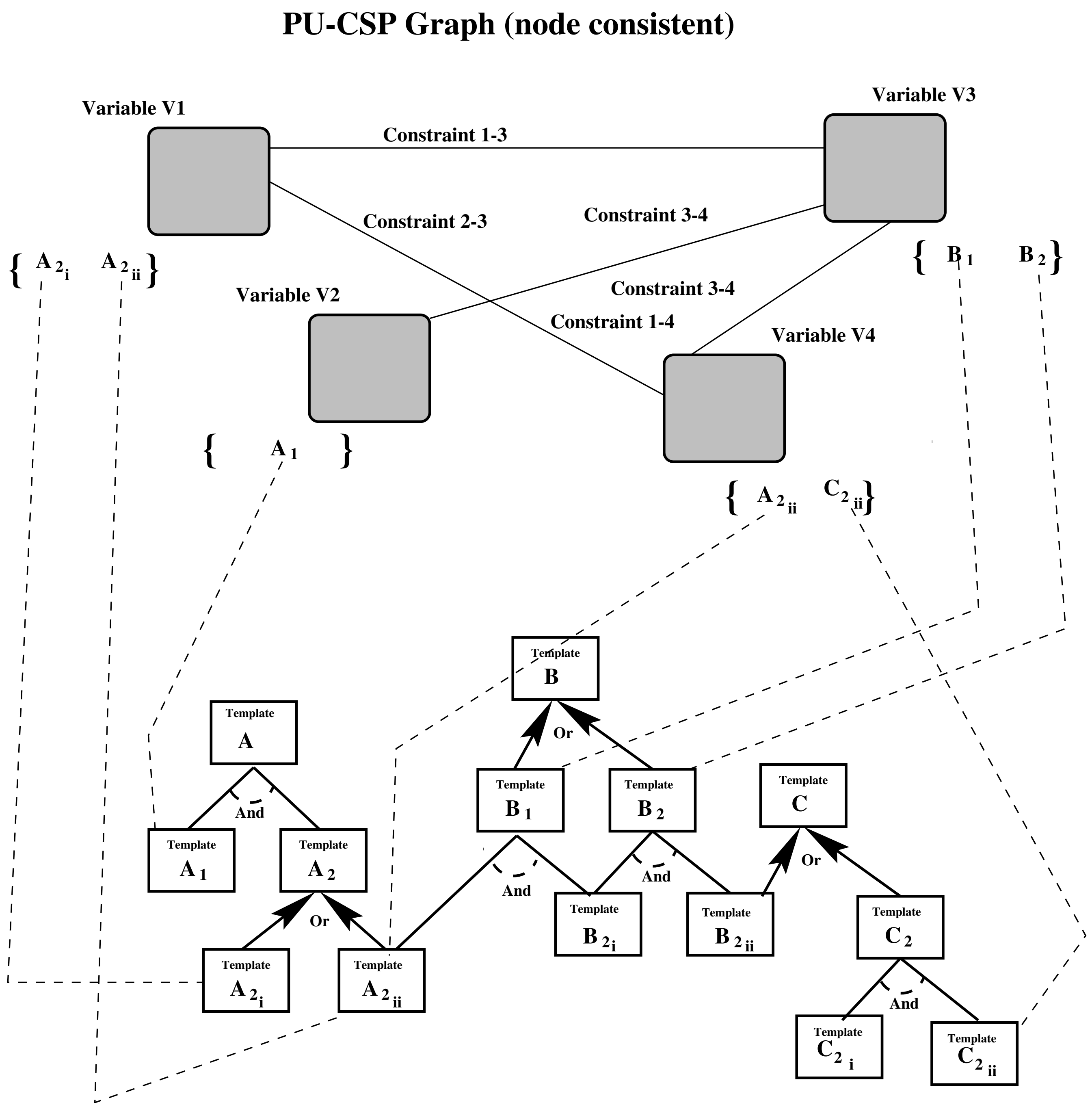

Figure 9: PUCSP Graph.

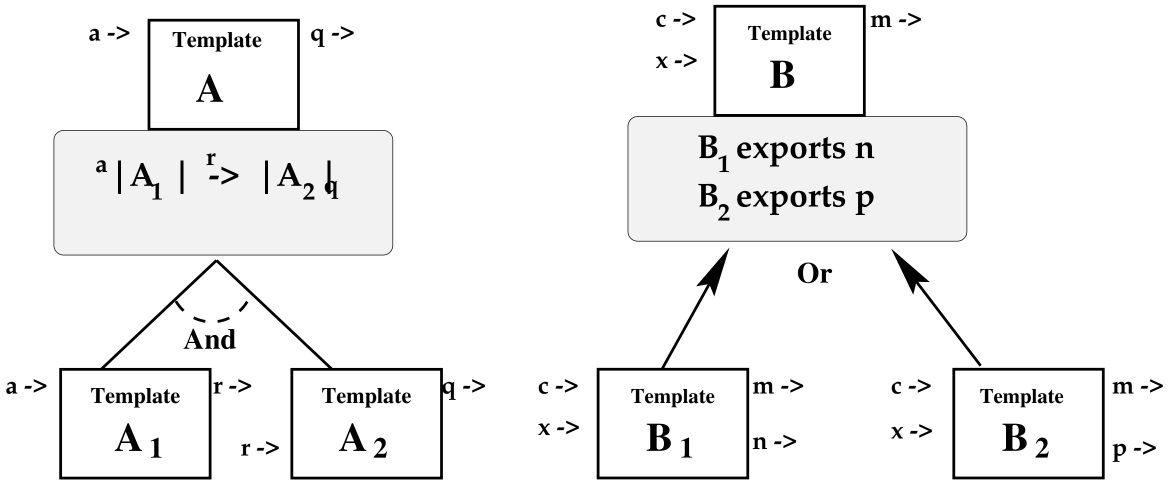

Figure 10: Library knowledge constraints.

CSP. Inference tests identify "related" program plan templates according to earlier component instantiation, and can be interpreted as a special kind of variable-ordering heuristic.

6 Program Template Matching as CSP: MAP-CSP

We have seen how PU-CSP resolves integration of "local" explanations of source code blocks. We represent the process of matching particular abstract program plans to our legacy source as the MAP-CSP. We view MAP-CSP as an integral part of the more ambitious understanding task. Successful matches "locally explain" certain program blocks, and these local solutions can then be exploited to restrict the larger PU-CSP.

A MAP-CSP or program template matching problem can be stated as follows: given a plan template with a number of elements and constraints among the elements, find all instances of the template in a source code. As an example, consider finding all instances of an abstract data type in a C program. Figure 11 is a **String** ADT plan template taken from a plan library. The ADT is described in terms of 5 features describing various key components of a string class. In addition, there are constraints among the different parts as well, such as the one that requires one component to go before another.

We could model this problem as a CSP. For the given plan template (or ADT), each feature is a variable in our MAP-CSP. The *domain range* consists of all possible source program statements. Variables here can have attributes such as (**print**,**for**) that may be seen as *constraints* on allowable assignment of program statements (values) to template features (variables). Other *constraints* are on the sharing of information among variables, and on the order in which template features or variable are expected to appear in legacy source.

A solution to the MAP-CSP consists of the set of all assignments of plan template features by source code statements, where each assignment must satisfy all constraints. As an example, consider the ADT of Table 1. When represented as a plan template as in Figure 11, the variables of the MAP-CSP are: $X_i, i = 1, \ldots, 5$. Initially the domain for each

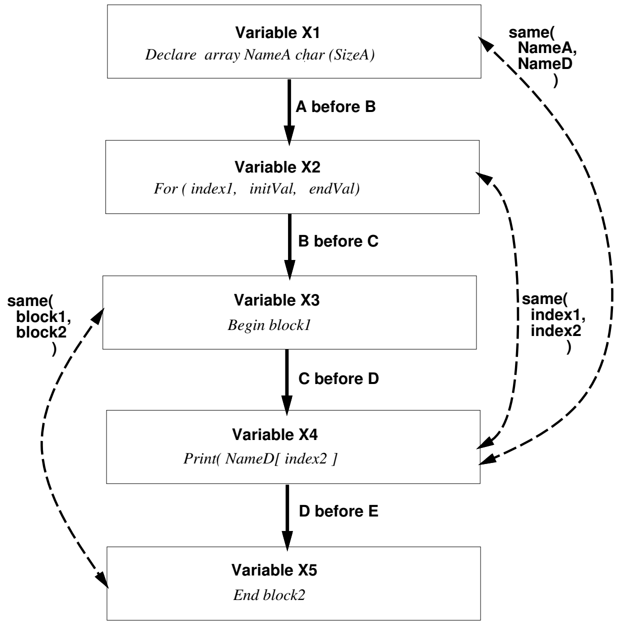

Figure 11: The **String** ADT in MAP-CSP.

variable ranges through all source statements in Figure 3. The constraints are as shown in the figure. The solution to this problem corresponds to the three alternative consistent assignments to the variables, one for each character string A, B and C, respectively. Thus, the solution to a MAP-CSP provides a mapping that *explains* the matched source statements as parts of an instance of the abstract program plan or ADT.

7 Empirical Results of MAP-CSP

In this section we present and discuss experiments which were intended to show the initial feasibility of the MAP-CSP representation and related algorithms in addressing the partial program understanding problem. For the purposes of this paper, this section is intended to provide the reader with an initial view of our experimental results. In subsequent work [39] we present extended results and description.

The format of the remainder of this section is as follows. First, we describe the program templates which will be instantiated in generated source data. Second, we explain how the modeled source data is generated for each set of experiments, and where the model for the data is obtained. Finally, we describe the results of identifying template instances utilizing several common algorithms for solving constraint problems.

While our ongoing research effort is directed towards the eventual demonstration of feasibility of both PU-CSP and MAP-CSP techniques in the domain of large commercial source libraries and legacy sources, that is not the focus of the experiments outlined in this chapter. Rather, we are interested in determining whether, with a minimum of *structural* constraint information, it is possible to utilize the CSP algorithm for MAP-CSP template recognition as input sources increase in size. For example, if the combinatorics of recognition stopped at 100 lines of code it would be necessary to re-consider this approach. However, if it is possible to scale to code on the order of a thousand lines, it is conceivable that MAP-CSP may be seen as a model or prototype of an integral sub-component of a future understanding toolset. In the future, such a model can be extended to take advantage of further structural source constraint information.

7.1 Program Plan Templates

Figure 12 shows an internal representation of our earlier example plan. This preserves the basic component and constraint representation, although the specific constraints vary from those used in the original systems. In particular, we approximate control and data-flow constraints using locality and containment constraints. In addition, we require a `same-name-p` constraint to capture the notion that a variable appeared in multiple places represents the same underlying entity, a notion that is implicit in **Decode**'s representation for plans. The component set elements consist of a general-component label, a general-component type, and component-identify constraint information (i.e. component q1-c has type "While", signifying that the template requires a loop component whose access is controlled by the boolean value "ResultA").

The **quilici-t1** (Figure 12) program plan template is derived directly from the `TRAVERSE-STRING` program plan utilized by Quilici in [22], and which we describe in detail in [39]. The results described in this paper will refer to experiments which locate instances of the **quilici-t1** template in source data. In [39] additional experiments are reported involving both smaller/simpler and larger/more complex templates.

The MAP-CSP template version of the **quilici-t1** template is composed of 9 primary components and 20 constraints.

Our experiments are concerned with all-instance template identification in large generated source instances. One primary concern (which we address in this paper) is how the relative size of the given source affects recognition performance. Another concern (addressed in [39]), is how the relative size of a program plan template (in terms of components and constraints) will affect the empirical performance of MAP-CSP.

7.2 Generated Example Sources

We have described one particular program plan template that we will be utilizing the MAP-CSP algorithm to search for in generated legacy examples. Before describing the process of legacy generation that we adopt, an obvious question is: "what does an instance of these program plans look like?". In answer to this question, Figure 13 presents a sample program fragment instance of the **quilici-t1** plan.

Our prime interest is the performance comparison of different approaches to program understanding in terms of the size of the programs being understood. In particular, our focus is on comparing the amount of effort expended (and consequently, time) in recognizing *all* instances of a single plan template as the source program is increased in size. To keep the focus on scale issues alone, our desire was to have programs of varying sizes available where those programs have the same relative distribution of different program entities (the

```
'( "quilici-t1"

   ( '' Component Set ''

     (q1-c While      (ResultA (boolean)))
     (q1-d Begin      (Block1 (block)))
     (q1-g Assign     (NameC (array (char))) (IndexC (int))
                           (ElemB (char)))
     (q1-e End        (Block2 (block)))
     (q1-i Increment  (IndexD (int)))
     (q1-a Decl       (NameA (array (char) (0 10000))))
     (q1-b Zero       (IndexA (int)))
     (q1-f Assign     (NameB (array (char))) (IndexB (int))
                             (ElemA (char)) )
     (q1-h Not-Equals (ElemC (char)) (NULL (char)) (ResultB (boolean)))   )

   ( ''Constraint Set''

     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
     (before-p    (q1-b q1-c))
     (before-p    (q1-a q1-b))
     (before-p    (q1-b q1-h))
     (before-p    (q1-d q1-e))
     (before-p    (q1-f q1-h))
     (before-p    (q1-g q1-i))
     (before-p    (q1-d q1-i))
     (before-p    (q1-i q1-e))
     (same-name-p (q1-c q1-h) (ResultA ResultB))
     (same-name-p (q1-f q1-h) (ElemA ElemC))
     (same-name-p (q1-a q1-f) (NameA NameB))
     (same-name-p (q1-a q1-g) (NameA NameC))
     (same-name-p (q1-b q1-f) (IndexA IndexB))
     (same-name-p (q1-b q1-g) (IndexA IndexC))
     (same-name-p (q1-b q1-i) (IndexA IndexD))  ))
```

Figure 12: CSP-based internal representation for plans.

```
Stmt Id  Line  Statement

(ADTQ1-A  100 (DECL ARRAY A CHAR 99))
(ADTQ1-B  200 (ZERO IDX))
(ADTQ1-C  300 (WHILE RESULT))
(ADTQ1-D  310 (BEGIN BLOCK1))
(ADTQ1-G  400 (ASSIGN A IDX ELEMB))
(ADTQ1-I  500 (INCREMENT IDX))
(ADTQ1-E  600 (END BLOCK1))
(ADTQ1-F  700 (ASSIGN A IDX ELEM))
(ADTQ1-H  800 (NOT-EQUALS ELEM NULL RESULT))
```

Figure 13: Instance of **quilici-t1** plan.

same percentage of loops, etc...) regardless of size. The test programs used as sources are automatically generated in the following way. An instance (or instances, depending on the experiment) of the program plan template is generated, and program statements are added randomly according to a pre-determined distribution of program statements. This distribution is derived directly from a cross-sectional study of student C programs undertaken by Quilici and described in [22].

Statement Type	Frequency	Percentage
While	1/22	4.5
Zero	1/22	4.5
For	1/22	4.5
Block	2/22	9.0
Increment	2/22	9.0
Not-Equals	2/22	9.0
Print	2/22	9.0
Assign	3/22	13.5
Decl	4/22	18.0
Check	4/22	18.0

Table 3: Program statement type distribution.

The experiments described here are based upon a distribution shown in Table 3. This distribution is derived directly from a cross-sectional study of student C programs undertaken by Quilici and described in [22]. In [39] we examine the effect on MAP-CSP of utilizing different distributions. When a variable was to be generated, it was generated with the following type distribution: array type (1/7), simple int (2/7), char (2/7), real (1/7), and boolean (1/7). If an array was generated, it was instantiated according to this type distribution: int (2/6), char (2/6), real (1/6) and boolean (1/6).

7.3 Problem Instances

Experiments with a given search strategy are performed based on the results of 10 MAP-CSP problem instances at each legacy source sample size. These 10 problem instances are generated according to the distribution described. Problem instances are created as follows: a particular program plan instance is generated from the template, including an assignment of line numbers for the instance according to the separation specified in the template. Legacy source statements are now generated according to the given distribution until a legacy program of appropriate size is generated. The statements are given line numbers randomly within the range from zero to the maximum line number specified in the template instance plus one hundred lines. Certain statement types (such as **Loop** with a corresponding **Begin** and **End**) require more than a single line in their generation. If a conflict occurs in which a new generated line number is already in use, a simple stepping algorithm selects the next available line number. If this algorithm hits the end of the allowed line range, the range is extended by one hundred additional lines.

As an example, utilizing the "Standard" distribution, a generated "program" containing one instance of the **quilici-t1** program plan together with 10 generated source statements is given in Figure 14. The template-related components may be identified in this (and subsequent) figures through the statement labels prefixed with "ADT". The initial template instance has 9 related component lines, and the remaining 12 added lines arise as a result of the insertion of a for-loop statement with 3 associated lines. Experiments of a particular size are generated at intervals of 50 legacy lines typically (although not in all cases). In such a case, the 10 examples at (say) size 250 would be graphed according to the average size of the 10 examples keeping in mind that each example has a slight variation depending on how many multiple-line insertions are made.

```
Stmt Id          Line      Statement

(|sit-gen5|     (0 88)    (ASSIGN FIRSTINT FIRSTINT))
(ADTQ1-A        (1 100)   (DECL ARRAY A CHAR 99))
(|sit-gen15|    (2 144)   (NOT-EQUALS FIRSTINT |var-name13| FIRSTBOOLEAN))
(ADTQ1-B        (3 200)   (ZERO IDX))
(|sit-gen1|     (4 289)   (CHECK FIRSTINT FIRSTCHAR))
(|sit-gen14|    (5 297)   (ASSIGN |var-name7| |var-name7|))
(ADTQ1-C        (6 300)   (WHILE RESULT))
(ADTQ1-D        (7 310)   (BEGIN BLOCK1))
(|sit-gen3|     (8 362)   (ASSIGN FIRSTARRAYI FIRSTINT FIRSTINT))
(ADTQ1-G        (9 400)   (ASSIGN A IDX ELEMB))
(ADTQ1-I        (10 500)  (INCREMENT IDX))
(|sit-gen6|     (11 574)  (DECL INT |var-name7|))
(|sit-gen2|     (12 584)  (ASSIGN FIRSTARRAYB FIRSTINT FIRSTBOOLEAN))
(ADTQ1-E        (13 600)  (END BLOCK1))
(|sit-gen0|     (14 632)  (CHECK FIRSTCHAR FIRSTREAL))
(|sit-gen8|     (15 682)  (FOR |var-name13| 14 7))
(|begin-sid11|  (16 683)  (BEGIN |block10|))
(|end-sid12|    (17 685)  (END |block10|))
(|other-sid9|   (18 686)  (ASSIGN FIRSTARRAYI |var-name13| |var-name13|))
(ADTQ1-F        (19 700)  (ASSIGN A IDX ELEM))
(|sit-gen4|     (20 765)  (CHECK FIRSTINT FIRSTINT))
(ADTQ1-H        (21 800)  (NOT-EQUALS ELEM NULL RESULT))
```

Figure 14: Instance of **quilici-t1** plan with 10 inserted statements.

7.4 Experimental Results

In this section we present a small sample of a larger range of experiments reported in [39] which are intended to show the feasibility of the MAP-CSP representation and related solution algorithms in relatively large (several thousand lines) problem instances.

The experimental results depicted here are based upon algorithms for constraint satisfaction described earlier. The solution algorithms referenced in the following figures include combinations of the following algorithms:

1. Standard BackTracking (BT, see Section 4.2).

2. Arc consistency propagation (AC-3, see Section 4.1).

3. Forward Checking with Dynamic Rearrangement (FCDR, see Section 4.2).

This list is not intended to be a complete set of solution strategies to constraint satisfaction strategies. Rather, these approaches represent a range of strategies that together are capable of capturing an initial subset of the heuristic strategies undertaken by previous program understanding researchers.

7.4.1 Detailed Individual Results

Single template instances
The following examples contain a single template instance in each generated legacy example. All of these examples are generated using the "Standard" Quilici distribution and make reference to identifying instances of the **quilici-t1** program plan template. The results are graphed showing a 95% confidence interval over the 10 sampled sources. All of these experimental instances were generated such that the inserted template was not destroyed, that is, the template was identified successfully in each case. In addition, for these examples no false solutions were identified. At the end of these *complete* searches, one may conclude that no other instance possibly exists that satisfies the template constraint set.

1. *Simple Backtracking* with no advance variable order, Figure 15. The experiment was terminated for legacy examples exceeding 400 lines. In fact, several individual cases failed to complete a total search of the given example in less than 20 cpu minutes, our arbitrary boundary. In particular, at 250 there was 1 failure, at 300 (1), at 350 (2) and at 400 (3).

2. *Simple Backtracking* with advance variable ordering, Figure 16. This experiment shows a rapidly increasing number of constraint checks as source example size increases. In particular, 25000 constraints are checked with an example size of approximately 725 source lines. Simple backtracking displays an extremely large variance between examples at the same source size. For instance, at 200 source insertions, one of the 10 cases required more than 500000 constraint checks to solve, while another at the same size required only 3400. For purposes of comparison between constraint checks with CPU time, it should be noted that the datapoint of approximately 950 source lines corresponds to slightly more than 100 CPU seconds on a shared SPARCserver 1000 running Allegro Common Lisp.

3. *AC-3 with Forward Checking and Dynamic Rearrangement* with advance variable ordering, Figure 18. For this experiment, the 25000 constraint check limit is surpassed at approximately 1200 source lines.

4. *Forward Checking and Dynamic Rearrangement* with advance variable ordering, Figure 17. This standard CSP solution strategy has typically performed well on a wide range of problems. The FCDR strategy essentially propagates constraints between each search assignment to a depth of one "look-ahead" variable. This experiment results in the checking of only 5274 constraints at 1500 source lines. A similar example for FCDR

shown in Figure 21 shows a result of 25000 constraint checks for an example source size of more than 3000 source lines. For purposes of comparison between constraint checks with CPU time, it should be noted that the last datapoint of approximately 1500 source lines corresponds to about 27 CPU seconds on a shared SPARCserver 1000 running Allegro Common Lisp.

5. Figure 19 graphs the median of the previous examples in a unified chart to show the relative performances. Note one line is extended for FCDR with advance variable ordering (2 solutions) so as to demonstrate the relative rate of increase for that heuristic. Figure 21 indicates how this particular heuristic scales in much larger examples.

6. Figure 20 demonstrates the relative utility of estimating effort through constraint checks as opposed to CPU time. In particular, we chart time on the X axis and constraint checks on the Y axis. We notice that initial overhead matters, however, only as a constant factor. The results are taken from the experiments utilizing FCDR (advance ordering) with the Standard template and Standard code distribution.

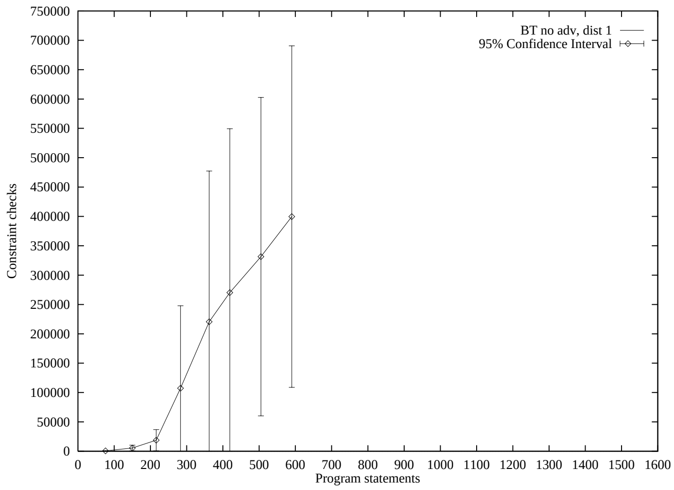

Figure 15: Standard BackTrack (95% conf. interval).

7.4.2 Comparative Results

The examples shown in the previous paragraphs for single template identification were conducted only up to a maximum of about 1500 legacy lines. Figure 21 shows the same (median) results charted in Figure 19 for a variety of search strategies, with the extension that the FCDR with advance variable ordering (median) tests have been extended to almost 6000 lines of code.

We wish to demonstrate that the MAP-CSP representation and algorithm is capable of providing all-instance results in moderately sized program slices. An efficient MAP-CSP algorithm could make the execution of the larger PU-CSP algorithm more feasible. In addition, the MAP-CSP algorithm for template matching could potentially be stand-alone

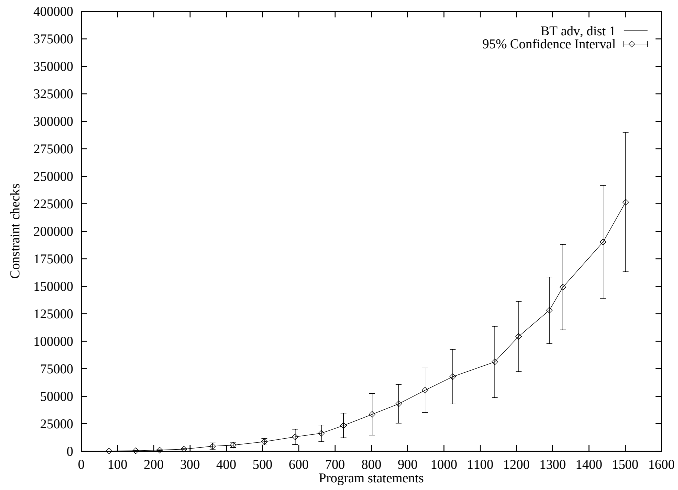

Figure 16: BackTrack, variable order (95% conf. interval).

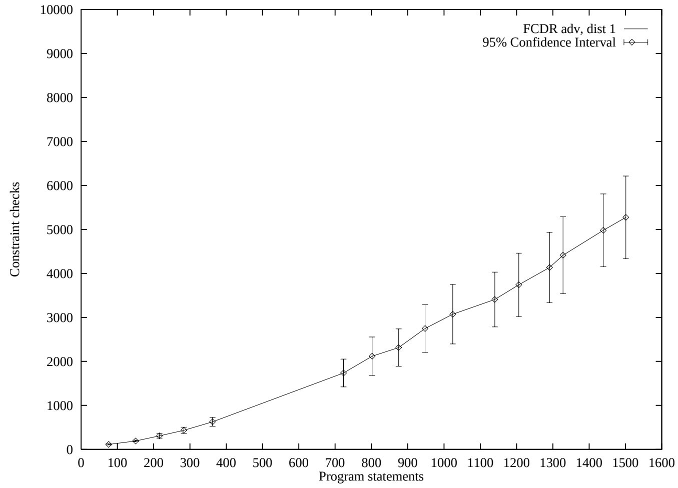

Figure 17: Forward Checking, DR (95% conf. interval).

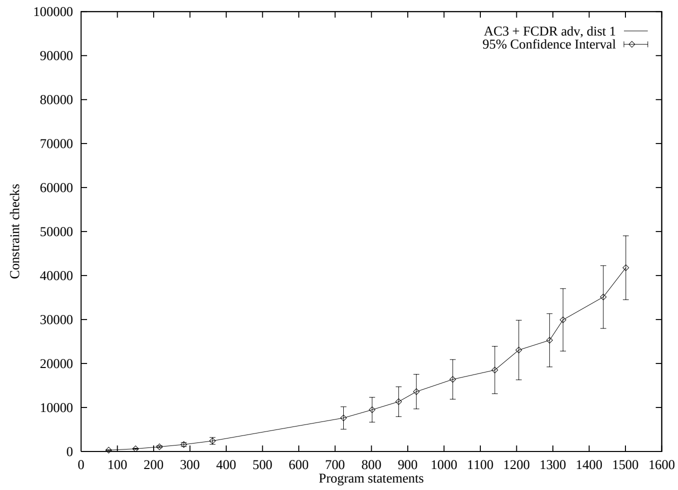

Figure 18: AC-3 with FCDR (95% conf. interval).

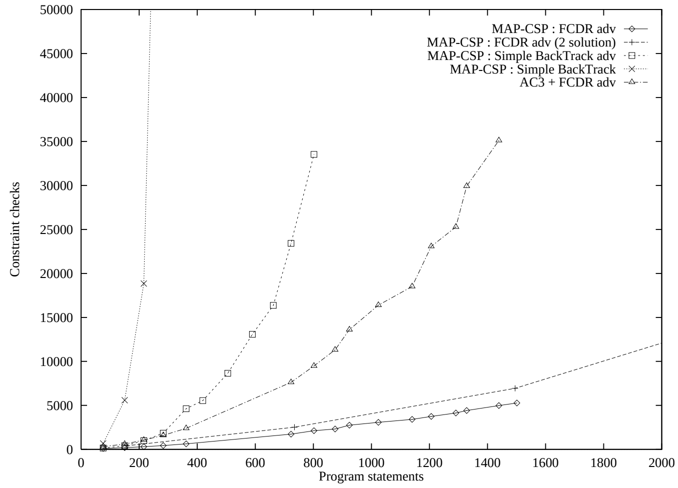

Figure 19: A range of strategies (medians graphed).

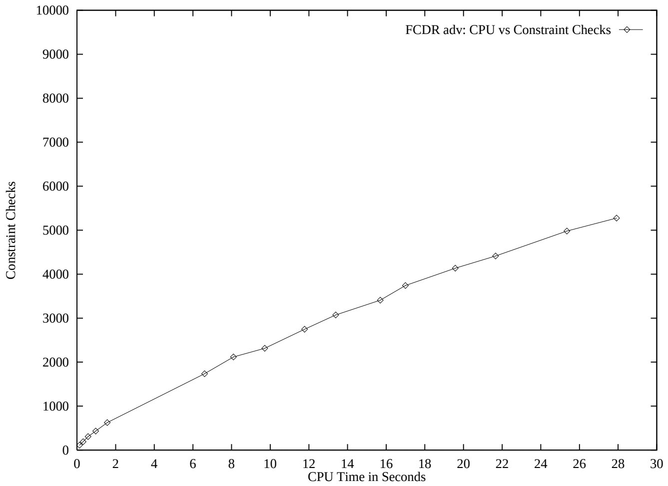

Figure 20: FCDR addv Constraints vs Time, Standard distribution.

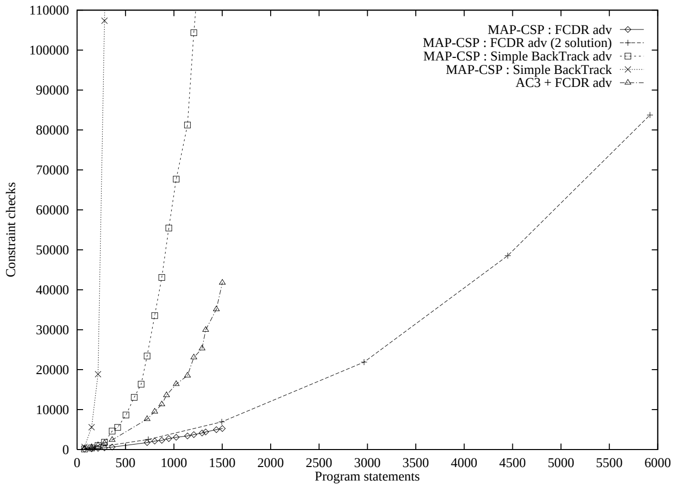

Figure 21: Extended results: strategy range.

as a tool for assisting in the identification of legacy source portions that may be replaced with existing source library objects.

Several observations can be made from our test results:

- Standard Backtracking exhibited very unstable performance in examples of the same size. As hoped, more intelligent strategies behaved in a more stable manner. Forward Checking was considerably more stable, and the applications using AC-3 in advance of search exhibited very small variance across test cases of similar size. Stability is an important factor in any application that may be used as part of an online or interactive tool. In addition, Standard Backtracking was unable to complete in less than 600 CPU seconds for source instances exceeding 500 program statements.

- For legacy source examples of up to approximately 600 lines of code, the FCDR strategies located all instances of the example program plan template in less than approximately 5 seconds of CPU time. In examples of up to 1500 lines of code, all instances were identified in approximately 35 seconds. In such prototyped (Lisp) near-real-time circumstances it would appear that a tool could be fashioned that could be called up to run as a background process supporting an expert working with some reasonably sized legacy code.

It is also clear the ratio of constraints evaluated to program statements is growing rapidly for our "standard distribution". This is problematic, but there are several reasons to remain optimistic. One is that our "simulated locality constraints" are much looser than the *structural constraints* such as control-flow and data-flow constraints found in real programs. We therefore predict that this curve will flatten significantly when experiments are undertaken in future work using programs that preserve the structural properties of real-world programs. This is because tighter constraints should reduce the size of the domain value sets, leading to a speedier solution to the CSP and fewer constraints to evaluate. It is an open question, however, just how much it will flatten and how much that will slow the rate of growth of the number of constraints evaluated.

The other reason to expect even better scalability using CSP is that, even if real-world structural constraints do not prevent the currently modeled plan recognition algorithm from going exponential, it appears from our curves that instances of individual plans can be recognized in programs in the 5,000 statement range in a reasonable amount of time. For instance, the 60,000 or so constraints we need to evaluate takes less than 25 seconds or so on a shared SPARCserver 1000 workstation. While 5000 statements is small compared to the sizes of real-world programs, it is within an order of magnitude of the size of modules in real-world programs or modules that have been created from legacy systems using semi-automatic techniques [21]. Other reverse and re-engineering toolsets such as **Rigi**[19] may be appropriately utilized in this process of source segmentation.

7.5 Context of this Work

This section has explored some scalability issues for a constraint-based approach to partial explanation using plan recognition. This, however, is only one part of the overall program understanding problem. Program understanding is often viewed as a three-step process:

Parsing: Turning the program into an annotated AST using standard parsing and flow analysis techniques.

Canonicalization: Simplifying this internal representation to minimize the number of different plans that must be in the library. A simple example is transforming all relational expressions so that they involve only the greater than operator and not the less than.

Plan recognition: Recognizing instances of each plan in a library of program plans.

This section has focused on preliminary experiments intending to support the scalability of one part of the plan recognition process: determining whether a given program plan is present based on the existence of the constrained plan components in the internal representation of the program. There are other aspects to this problem, that we have not yet addressed, including how to decide which plans to try to locate within a given program, and in which order to try those plans. Section 5 introduced these integrative issues, and [39] expands them in detail.

Our experiments have several important implications. One is that it may well be necessary to have a modularization step that precedes the plan recognition process, where this step breaks the program into pieces of whatever size the program understanding algorithm can comfortably handle before the combinatorics become problematic. In fact, this is precisely the point of the PU-CSP stage for integrating these partial solution stages. Some work on semi-automatic modularization of COBOL program has already been done that has demonstrated that large COBOL programs can be broken into modules of 25,000 or so statements [21]. This is only a factor of 5 larger that the point that the CSP approach can comfortably handle, which makes it appear worthwhile to determine whether those techniques can be extended to break programs down into even smaller modules.

In addition, even if we successfully recognize plans at the module level, there also needs to be a mechanism for combining this modular understanding that needs to follow the plan recognition process. It's an open question how we accomplish this task to come up with an understanding for a program as a whole, especially if the library is incomplete and we have only partial understanding of what a module does.

Finally, our success in using CSPs in the local or MAP-CSP understanding process suggests that perhaps they can be applied to other related tasks, such as selecting plans to recognize, or as part of the canonicalization process. However, it is an open and interesting research question how to do so.

7.6 Looking Ahead

Our results may be thought of as some initial, overdue, data points in a progress report on the state of the art of program understanding. In particular, the specific amount of work done by the CSP recognition algorithm can be reduced, perhaps significantly, by moving to real control-flow and data-flow constraints, an experiment we are intending to set up in a future extension to this work. This may well mean that significantly larger programs can be successfully understood. In addition, the relative amount of work done by the algorithm may increase rapidly as we move toward exploring larger and larger plans, rather than slowly as it had with our first few plans. This may mean that there is an practical upper bound on the size of individual plans that can be efficiently recognized in a program of a particular size. If our initial empirical results hold up, they suggest that automatically modularizing large programs and combining modular understanding are several important areas of future research. Our PU-CSP model addresses integration of these modularized partial explanations.

Our hope is that this work will spur others working in the area of program plan recognition to do one of two things: either map their understanding algorithms into the CSP framework so that others may easily compare their performance with our CSP approach, or to provide data on the performance of their program understanding algorithms as programs grow in size. This step is crucial to move beyond the understanding of "toy" programs and into the world of being a useful aid in the re-engineering of real legacy systems.

8 Conclusions

In this paper we have constructed a general representation of the program understanding task as a constraint satisfaction problem. Two versions of the task are identified: one is to find all instances of a given program plan template in a source code, and the other is to construct or verify an explanation of the source code in terms of a program plan library. In addition, we have modeled various search heuristics for program understanding as instances of a generic CSP search algorithm. We believe that the algorithm subsumes the previously proposed methods for the same problem, and can be systematically studied on a spectrum of heuristics.

We have also implemented the all-instances template matching problem, MAP-CSP and demonstrated that MAP-CSP can be solved for problems of non-trivial size using intelligent backtracking and constraint propagation within a reasonably stable and reasonably short time period. MAP-CSP has potential application both as a stand-alone tool for legacy code reduction and as a key component within the program understanding task.

We summarize some of the advantages of our approach below.

Scalability Our empirical results demonstrated that the MAP-CSP problem can be scaled up for legacy code of moderate sizes - 6000 lines of generated code in some cases. This efficiency gain is achieved by viewing the recognition problem as constraint satisfaction, and applying known constraint satisfaction algorithms. This focus-range is rapidly approaching that achieved in automatic-modularization efforts in COBOL programs. In our experiments, we haven't utilized the full range of constraints inherent in a program source code, such as those derived from program parsing, a technique employed by Kozaczynski & Ning[11] and Wills[37]. More extensive consideration is given to the specific use of these constraints in [39], and is a part of our continuing research. We expect the empirical results to improve further with use of these constraints.

Usability We envision our system as one part of a programmer's assistant toolset. The basis of such a toolset would undoubtedly be a visual reverse/re-engineering platform such as suggested by Muller's **Rigi**. For the MAP-CSP problem, a programmer could use the system to identify abstract program plans in legacy fragments of as many as 1000 lines of code in near-real-time, and can apply the system in batch-mode to much larger programs.

We have been involved 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 5% of the code in terms of existing software libraries would greatly benefit software maintainers.

In other work [39] we report on the implementation of a hierarchical search algorithm for PU-CSP. We expect to see similar effective results from constraining the search with hierarchical plan knowledge, particularly when this algorithm is fully integrated with the MAP-CSP solutions.

Acknowledgments

We thank Alex Quilici and Jim Ning for their insight and comments and Grant Weddell for many helpful discussions. 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 (ITRC).

References

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

[2] Sandra Carberry. Incorporating default inferences into plan recognition. *Proceedings of the 8th AAAI*, 1:471–478, 1990.

[3] Martin C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[4] E.C. Freuder. A sufficient condition of backtrack-free search. *Journal of the ACM*, 29(1):23–32, 1982.

[5] R.M. Haralick and G.L Elliott. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[6] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up problem-solving by abstraction: A graph-oriented approach. Technical report TR-95-07, University of Ottawa, March 1995.

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

[8] Rick Kazman and Marcus Burth. Asssessing architectural complexity. URL: `http://www.cgl.uwaterloo.ca/~rnkazman/assessing.ps`, 1997.

[9] K. Kontogiannis, R. DeMori, R. Bernstein, and M. Merlo. Localization of design concepts in legacy systems. *Proceedings of the International Conference on Software Maintenance*, pages 414–423, September 1994.

[10] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE)*, pages 68–73, August 1995.

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

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

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

[14] Alan Mackworth, Jan Mulder, and William Havens. Hierarchial arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:188–126, 1985.

[15] Steve Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[16] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[17] H. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsytem structure identification. *Journal of Software Maintenance*, 5(4):181–204, December 1993.

[18] H. Müller, M. Tilley, M.A. Orgun, B. Corrie, and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection modules. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '92)*, ACM Software Engineering Notes, pages 88–98, December 1992.

[19] H. Müller, 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.

[20] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[21] P. Newcomb and L. Markosian. Automating the modularization of large COBOL programs: Application of an enabling technology for re-engineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 222–230, 1993.

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

[23] Alex Quilici. Toward practical automated program understanding. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE-95)*, August 1995. In conjunction with the Fourteenth Int'l Joint Conference on Artificial Intelligence.

[24] Alex Quilici and David Chin. A cooperative program understanding environment. In *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*, pages 125–132, Monterey, CA, 1994.

[25] 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.

[26] Alex Quilici and Steven Woods. Toward a constraint-satisfaction framework for evaluating program-understanding algorithms. *Journal of Automated Software Engineering*, 3(4):271–289, July 1997.

[27] Alex Quilici, Steven Woods, and Yongjun Zhang. Some new experiements in program plan recognition. In *Proceedings of the Fourth Working Conference on Reverse-Engineering*. IEEE Computer Society Press, 1997.

[28] Alex Quilici, Qiang Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. URL: `http://spectra.eng.hawaii.edu/~sgwoods/Sub/AuSE-prpu-paper.html`, submitted for publication, 1997.

[29] C. Rich and R.C. Waters. The Programmer's Apprentice: A research overview. *IEEE Comput.*, 21(11):10–25, 1988.

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

[31] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. The interleaving problem in program understanding. In *Proceedings of the Second Working Conference on Reverse-Engineering*, pages 166–175, 10662 Los Vaqueros Circle, Los Alamitos CA 90720-1264, July 1995. IEEE Computer Society Press.

[32] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 24-28 Oval Road, London England, NW1 7DX, 1993.

[33] Peter van Beek, Robin Cohen, and Ken Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence*, 9(3), 1993.

[34] Arie van Deursen, Steven Woods, and Alex Quilici. Program plan recognition for year 2000 tools. In *Proceedings of the Fourth Working Conference on Reverse-Engineering*. IEEE Computer Society Press, 1997.

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

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

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

[38] Steven Woods. A method of interactive recognition of spatially defined model deployment templates using abstraction. In *Proceedings of the Knowledge Based Systems and Robotics Workshop*, pages 665–675. Government of Canada, November 1993.

[39] Steven Woods. *A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering.* PhD thesis, University of Waterloo, July 1996.

[40] Steven Woods and Alex Quilici. A constraint-satisfaction framework for evaluating program-understanding algorithms. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC-96)*, Berlin, Germany, March 1996.

[41] Steven Woods and Alex Quilici. Some experiments toward understanding how program plan recognition algorithms scale. In *Proceedings of the Third Working Conference on Reverse-Engineering*, November 1996.

[42] Steven Woods, Alex Quilici, and Qiang Yang. Program understanding : A constraint satisfaction modeling framework; understanding as plan recognition. Technical Report CS 95-52, University of Waterloo, Department of Computer Science, 1995.

[43] Steven Woods, Alex Quilici, and Qiang Yang. *Constraint-based Design Recovery for Software Reengineering: Theory and Experiments.* Kluwer Academic Publishers, University of Hawaii at Manoa (USA), Simon Fraser University (Canada), 1997. URL: `http://www.wkap.com/`.

[44] 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.

[45] 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. IEEE Computer Society Press, July 1995. Also appears in the *Proceedings of the 1995 Second Working Conference on Reverse Engineering (WCRE).*

[46] Steven Woods and Qiang Yang. Approaching the program understanding problem: Analysis and a heuristic solution. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996. IEEE Computer Society Press.

[47] Steven Woods and Qiang Yang. Approaching the program understanding problem: Analysis and a heuristic solution. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996. IEEE Computer Society Press.

[48] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6, 1990.

[49] Qiang Yang. A theory of conflict resolution in planning. *Artificial Intelligence*, 58(1-3):361–392, 1992. Special Issue on Constraint-directed Reasoning.

[50] Yongjun Zhang. Scalability experiments in applying constraint-based program understanding algorithms to real-world programs. Masters thesis, University of Hawaii at Manoa, Department of Electrical Engineering, 1997.