Program Understanding as Constraint Satisfaction

Steven Woods   Qiang Yang

Department of Computer Science
University of Waterloo
Waterloo, ONTARIO N2L 3G1
{sgwoods,qyang}@logos.uwaterloo.ca

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 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.* 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 agent utilizes this mapping in further inferring the possible high-level goals of the program. The expert may exploit such a mapping or partial mapping in order to: (1) translate the program into the source code of another programming language (e.g., C to C++); (2) to assist in debugging source code logic problems; or (3) to replace the understood code portions with generic code from a pre-existing library. In many real-world circumstances, a reduction in the size of a large legacy source code by only a small percentage could result in a substantial reduction of the overall maintenance cost.

In artificial intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition [7, 1, 2, 16]. In many of these works, existing human knowledge in a particular domain is represented as hierarchies of plans that describe relevant actions and goals. Given such a hierarchy, and 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 amount of search.

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[14], Kozaczynski and Ning[8], Rich and Waters[15] and Wills[18, 19]. To some extent, all are aimed at improving the effectiveness of the mapping process through heuristic knowledge.

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[19] who presents empirical results 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)*[10]. 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[8, 19]. 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.

2 Program Understanding as a CSP

In this section, we present a representation of program understanding problem as a CSP. To begin with, we give an overview of the program understanding problem.

2.1 The Program Understanding Problem

Consider the C program outlined on the left hand side of Figure 1. 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.

To understand this program, one might use as a basis a library of program plans as shown in Figure 3 which represents previously compiled knowledge about program composition within a particular domain. Figure 2 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 1.

```
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]); }   }
```

Figure 2: Example abstract data type.

Given the legacy source code on the left side of Figure 1, we would like to *understand* or *explain* some

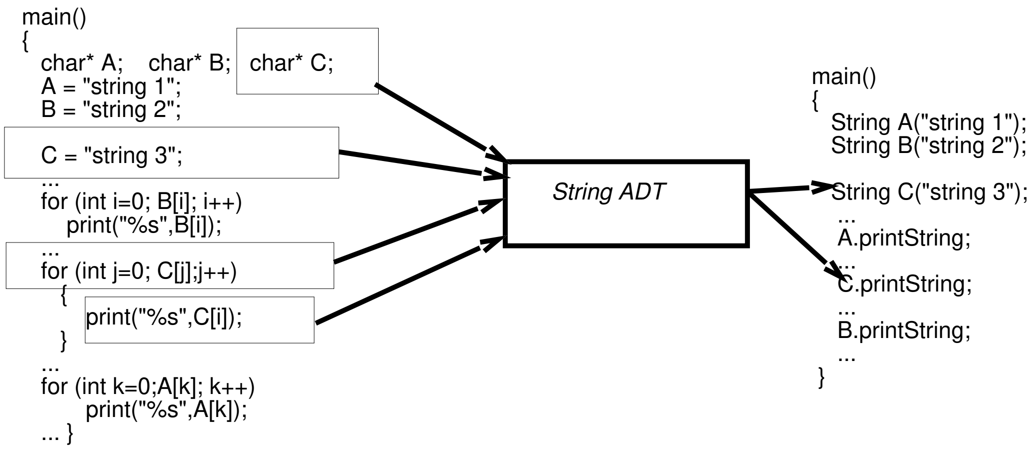

Figure 1: C legacy code mapped as String ADT instance to C++ code.

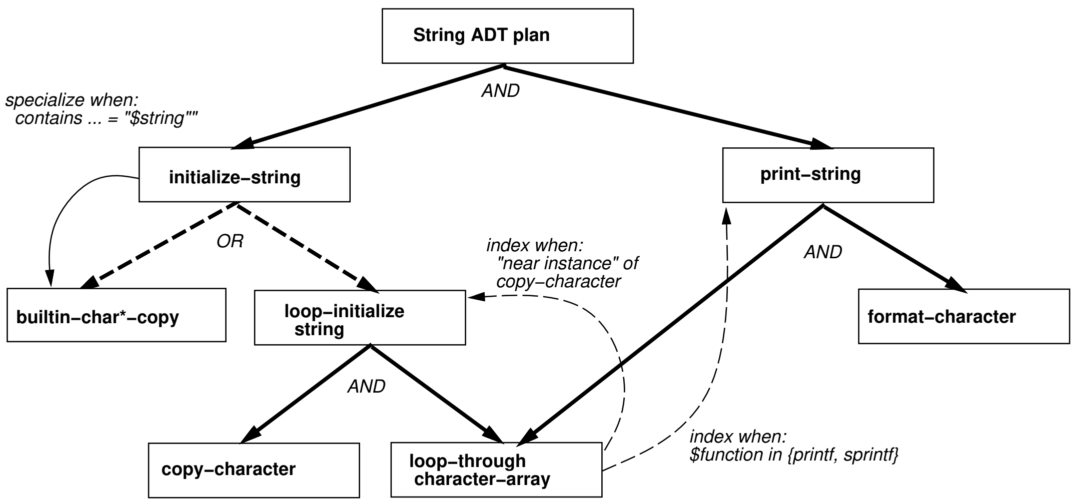

Figure 3: **String** ADT within a hierarchical program plan library.

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 1 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.

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.

2.2 Selected related work

Quilici's memory based method

Quilici[14] constructs an explicit library of programming plan templates, complete with indexing ability, which can quickly associate a particular 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 3. 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 3 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 3, 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 susbtantial 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 between candidate unexplained components, (2) choosing between multiple initial plan assignments for a component, (3) choosing between 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.*

Wills graph parsing method

Wills[15, 18, 19] 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 so as 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. This 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.

2.3 A CSP representation

A Constraint Satisfaction Problem[1] typically consists of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints amongst the variables which restrict domain value assignments. A solution of a CSP is a

[1]See [9] for an accessible and detailed treatment of Constraint Satisfaction Problems.

set of domain value-to-variable assignments such that all inter-variable constraints are satisfied. For example, the map-coloring problem may be interpreted as a CSP in which every country on the map is a variable, every variable ranges over all available colors, and the restriction that no two adjoining countries may be colored the same is represented as constraints between each neighboring pair of countries. Depending on the problem domain, one may be interested in *any one* solution (any satisfying coloring), any solution satisfying some "goodness" measure (a coloring that meets some tasteful color combinations), or even *all solutions* (all satisfying colorings).

As stated earlier, we formulate the program understanding problem in terms of two related CSPs, PU-CSP and MAP-CSP, which are explained in turn in the following sections.

Program Understanding as 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 domain ranges* correspond to all possible explanations of an individual source code block. As an example, consider the legacy code program statements of Figure 1 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 3.

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 1, 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 1 were instantiated to program plan **builtin-char*-copy** as shown in Figure 3, 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 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. We elaborate these observations further in Section 3.

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 4 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

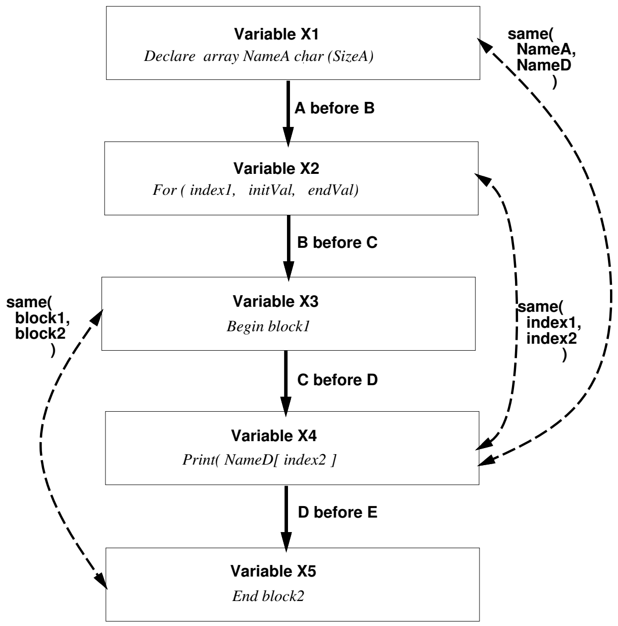

Figure 4: The **String** ADT in MAP-CSP.

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 Figure 2. When represented as a plan template as in Figure 4, the variables of the MAP-CSP are: $X_i, i = 1, \ldots, 5$. Initially the domain for each variable ranges through all source statements in Figure 1. 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.

3 Search Algorithms

We have explained program understanding as a form of CSP. Solving a CSP requires *constraint propagation* which attempts to systematically reduce variable domain ranges through the application of constraints between variables, or *heuristic backtracking* methods which repeatedly select variables and instantiations and reduces remaining variable domains according to existing constraints. A thorough examination of these techniques can be found in [13] and [9]. During a backtracking search, each variable instantiation is interpreted as extending the current understanding of a legacy program one step further.

Using the CSP representation, we can also consider a more systematic study of different search algorithms. Figure 5 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* are the determination of variables and domain values before the search starts. It 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 domain ranges locally or globally within the CSP problem graph. Existing algorithms include AC-3[10], AC-4[12], AC-5[17], 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 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 so as 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.

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.
3. Solution = NULL
4. [**Variable Selection**] Select and remove a variable X from V
5. [**Value Selection**] Select and remove a value of X from $Dom(X)$.
   The value must be consistent with all assignments in Solution.
6. [**In-search Propagation**] Apply a subset of constraints to V.
7. [**Backtrack Point Selection**] Backtrack if any $Dom(X)$ in V becomes empty.
8. [**Solution Evaluation**] If V is empty, exit with Solution (if all-solution, continue);
else, goto Step 4.

Figure 5: Generic CSP Search Algorithm.

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 [13] 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, if there exists particular measures of adequacy or *soft, preferential* constraints that may have been relaxed during search, such measures may be applied here.

[2]These and other intelligent backtracking algorithms are described in detail by Nadel in[13].

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*[11]. 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 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 can 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 will 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 domain ranges, 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, 20].

4 Empirical Results of MAP-CSP

In this section we present and discuss experiments which are intended to show the feasibility of the MAP-CSP representation and related algorithms.

In Figure 4 a CSP is described for the **String** ADT. This CSP contains 5 variables each corresponding to a part of the program plans contained in the ADT in Figure 2. The domain values are made up of source program statement blocks. For this test problem, there are 4 precedence constraints amongst the template variables, along with one additional constraint that the **begin** block corresponding to the **for** statement be within 15 program lines (the number 15 is arbitrarily chosen).

A test case is produced by instantiating 3 instances of the program template in a sample source code, and by adding some variable amount of additional program statements as "noise" around the template instances[3]. For example, the legacy source example shown in Figure 1 contains three separate instances of the **String** ADT of Figure 4.

Our experiments are undertaken using a specific version of the generic CSP search algorithm. This algorithm has been implemented in Common Lisp on a SPARCserver 470 workstation with four different heuristic configurations:

- Standard BackTracking
- Forward Checking with Dynamic Rearrangement
- AC-3 in advance of Standard BackTracking
- AC-3 in advance of Forward Checking

Each configuration has been applied to legacy sources ranging in size from 50 source lines to 1000 source lines, in 50 source line increments. Each increment was tested with 10 different random problem instances.

The MAP-CSP experiments are detailed in Figure 6[4] for *Standard BackTracking*, in Figure 7 for Forward Checking with Dynamic Rearrangement, in Figure 8 for AC-3 constraint propagation in advance of Standard BackTracking, and in Figure 9 for AC-3 in advance of Forward Checking. Each figure shows the number of CPU seconds required to find all template instances, with a 95% confidence interval charted.

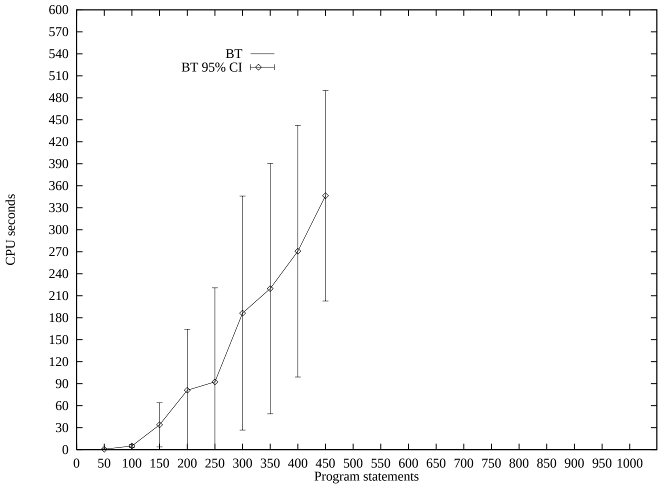

Figure 6: Standard BackTrack (95% conf. interval).

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 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:

[3] The experiments presented in this paper are undertaken using artificially generated, potentially nonsensical, legacy code. Our ongoing research effort is focused on testing the integrated PU-CSP and MAP-CSP techniques to large existing commercial source libraries.

[4] If more than 2/10 of the experiments for a particular level of program statements did not complete in 600 CPU seconds, those results are not charted. This occurs in the results shown in Figures 6 and 8.

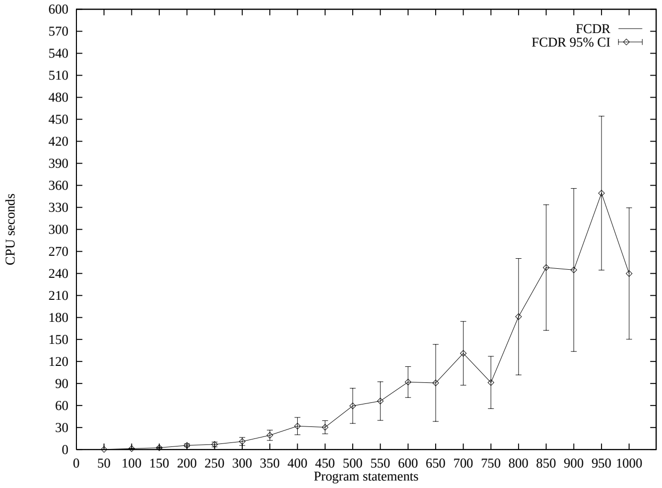

Figure 7: Forward Checking, DR (95% conf. interval).

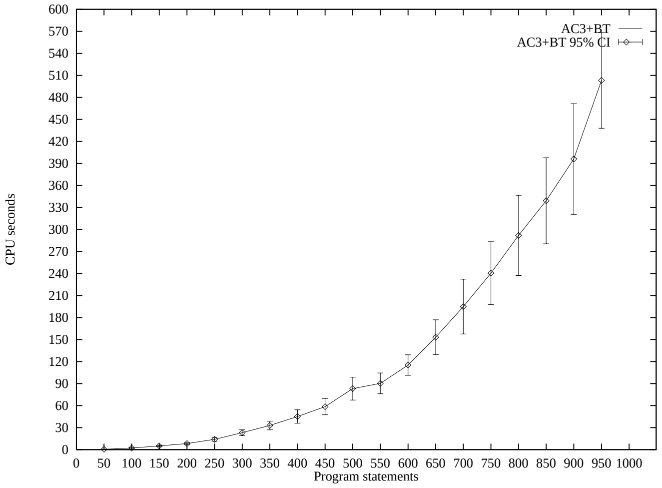

Figure 8: AC-3 with BT (95% conf. interval).

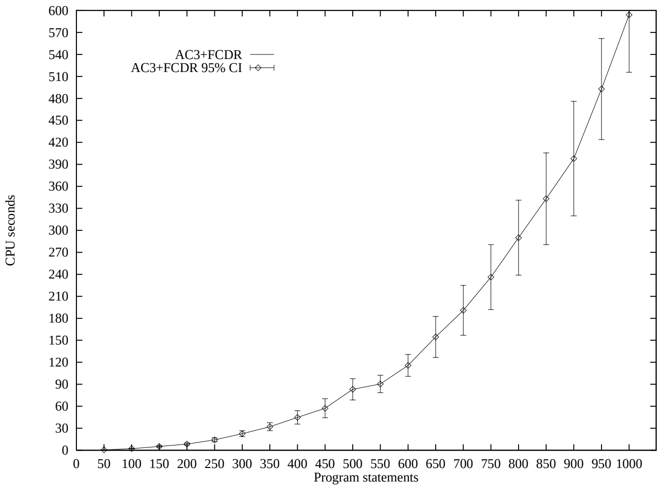

Figure 9: AC-3 with FC, DR (95% conf. interval).

- 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 500 lines of code, the intelligent strategies located all instances of the ADT in approximately one minute of CPU time. In examples of up to 300 lines of code, all instances were identified in approximately 30 seconds. In such 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 legacy code.

- In experiments where the number of source lines exceeded 200, the appearance of *false solutions*[5] started to become apparent. These solutions arise through combinations of actual template instance components and nearby program statements that meet all of the constraints of the ADT. However, the number of false solutions never exceeded 10 in programs of 900 or less lines, and rarely exceeded 5 in smaller sources. These results suggest that *either* our template specifications need to be tightened somewhat so as to exclude these false solutions, or the system should be capable of interacting with an expert who may verify solutions before they are adopted. It is important to note that in the solution of the larger PU-CSP it is expected that these false solutions will be identified and discarded, primarily on the strength of *knowledge* constraint restrictions.

5 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

[5] A "false solution" is a satisfying assignment of template variables to program parts such that the template constraints are satisfied, however, the found mapping is in fact not an actual instance of the program template. They arise as a result of overly abstracted template specifications.

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. 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.

Acknowledgments

We thank 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 Institute for Computer Research (ICR).

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

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

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

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

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

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

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

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

[16] Peter van Beek, Robin Cohen, and Ken Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. advance version of journal paper, 1994.

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

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

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

[20] 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.