Ph.D. Second Stage
Thesis Proposal

A constraint-based approach to program plan recognition in software reverse engineering

We investigate the application of artificial intelligence plan recognition and constraint satisfaction algorithms to perform reverse-engineering and program understanding based on known programming knowledge such as abstract data types and program schemas. We intend to demonstrate the feasibility of such an approach by performing empirical systematic tests on real legacy code.

Steven Woods
University of Waterloo
Waterloo, Canada

February 14, 1995

Abstract

The process of understanding a source code in a high-level programming language involves complex computation. Given a library of program plan templates, understanding the code corresponds to building a mapping from parts of the source code to a particular program plan. This mapping could be used to *reverse engineer* a legacy code, to facilitate *software reuse*, and to enable a program to be *translated* to another program in a more modern programming language, in a semi-automatic fashion. In this proposal we present a unique model of program understanding using constraint satisfaction. In this model we compose a global picture of the source program code by transforming knowledge about the problem domain and the program itself into constraints to be handled. We present an initial empirical study which demonstrates the feasibility of one major implemented component of our model over a range of known search algorithms. We outline a proposal to extend and refine our model, and to conduct a systematic empirical study based on real legacy source code. This study will range over known search and constraint propagation methods for solving constraint satisfaction problems. 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 can be easily compared. Another advantage is the improvement in search efficiency using various heuristic techniques in constraint satisfaction.

Contents

Chapter 1

Introduction

1.1 Overview

A key aspect of human intelligence is to successfully interpret an explicit representation of knowledge presented by another agent. In software engineering, this skill is often applied to the task of *program understanding.* An expert agent's understanding of a given source program can be considered a *mapping* from an existing mental model to the components of the source code. This mapping assists the agent in inferring the program's high-level goals. By successfully performing this mapping, an agent is then more able to translate the program into the source code of another programming language (e.g., C to C++), to debug what might be wrong with the original code at a logic level, or to replace the understood code with generic code from a pre-existing library. 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.

In Figure 1.1 an expert attempts to map a piece of legacy code to an existing mental model as part of the task of understanding code.

1.2 AI Plan Recognition

In artificial intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition (Allen & Perrault 1978, Allen & Perrault 1980, Kautz & Allen 1986, Carberry

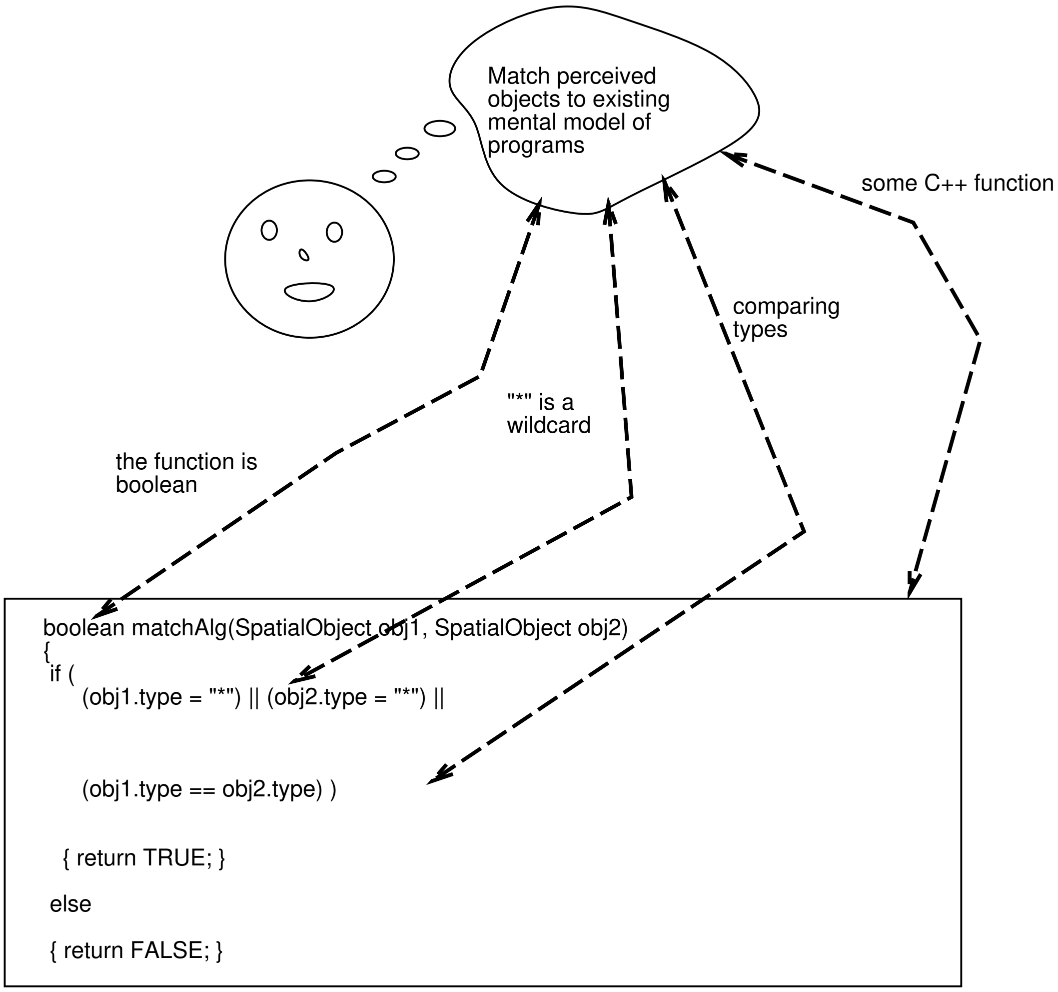

Figure 1.1: Mapping legacy source code to a mental model

1985, Carberry 1988, Carberry 1990, Mooney, Carberry & McCoy 1991, van Beek, Cohen & Schmidt 1994). In this area, 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 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.

In Figure 1.2 we show an excerpt of a program plan library representing a particular model of domain knowledge for object matching. A mapping is shown between components of a small piece of legacy source code and the model. This mapping represents one possible interpretation of those components which suggests a possible inference that this piece of code belongs to the "Type-Match" class.

1.3 Software Reverse Engineering

Work reported in Rich & Waters (1990) characterizes the attempt to recognize program plans by constructing a mapping between legacy source code and a pre-existing library of program plans or clichés. Other researchers have sought to extend this approach, or pursue parallel approaches in program understanding in an effort to realize a recognition system that is capable of efficiently identifying at least partial mappings in certain well understood domains. One notable recent effort of this type is reported in Quilici (1994). In this work, an explicit library of programming plan templates is constructed, 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. In a related work, Grimson (1990) demonstrates the need for both *indexing* and *selection* (grouping of related elements) in partial understanding.

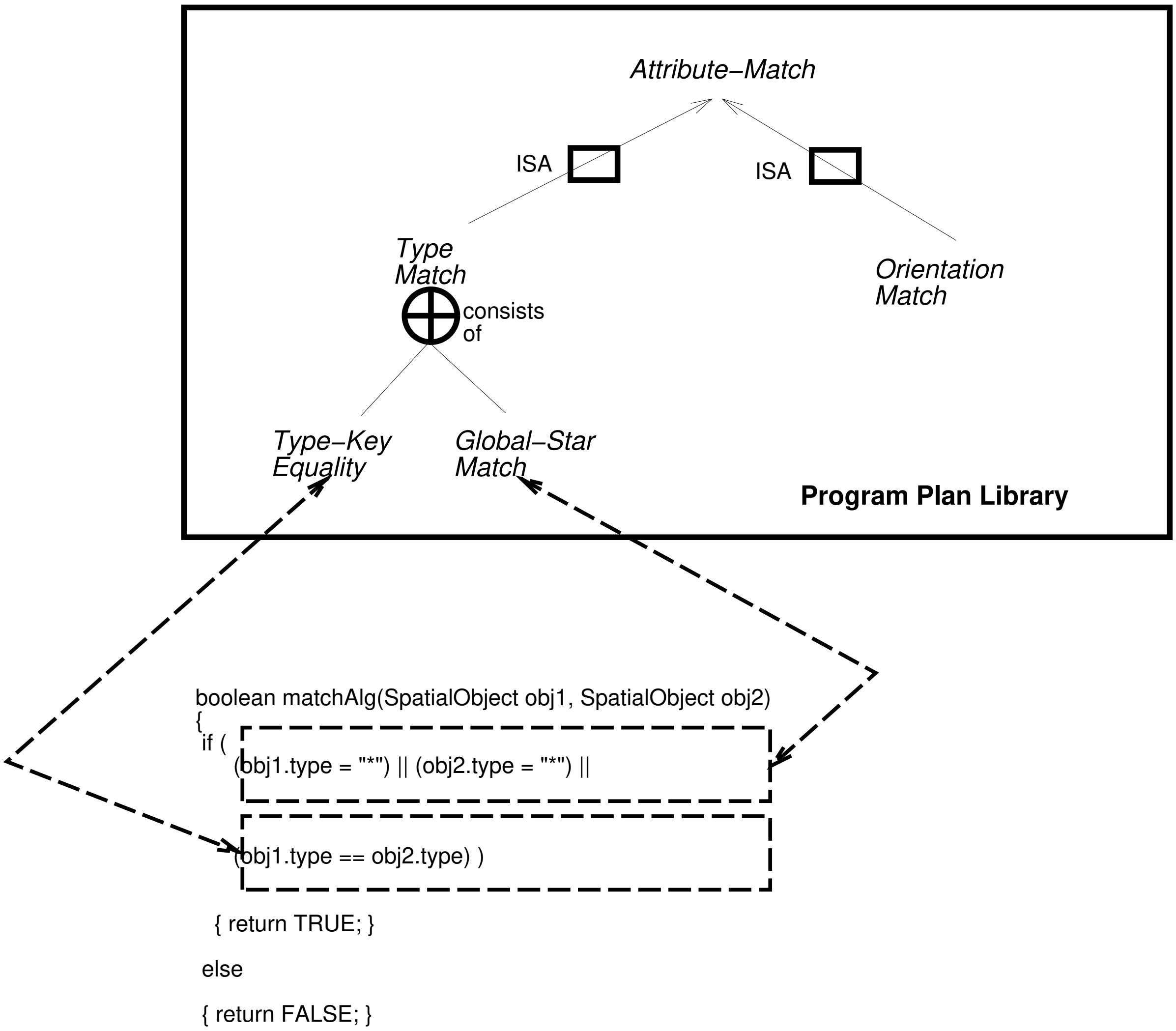

Figure 1.2: Mapping between Program Plan Library and Source Code

Kozaczynski & Ning (1994) present a method of program understanding based on the identification of *abstract concepts.* These concepts are defined within a program plan library much as in Quilici, with each concept including attribute information. Each concept is composed of other less abstract subconcepts and constraints on and among concepts. Like Quilici, Kozaczynski & Ning (1994) specify not only the concepts one would like to recognize, but *how* one should recognize them. Constraints specified include semantic legacy code relationships such as *control-flow*, *data-flow*, and *calling relationships.* This model builds abstract concepts *top-down* by invoking specific "larger concept recognition methods" based on the identification of certain abstract syntax elements. At a very low-level of abstraction, Paul & Prakash (1993) describe a similar recognition approach based upon matching syntactic clichés with source constructs in legacy abstract syntax trees annotated with semantic relations[1].

Wills (Rich & Waters 1990, Wills 1990, Wills 1992*a*, Wills 1992*b*) 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 (1992*b*) 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.

1.4 Primary contributions

The work we propose has the following potential contributions:

1. *Representation*

[1] Paul & Prakash (1993) refer to these relations as *non-structural.*

(a) *Representing program understanding*

The approaches of Kozaczynski & Ning (1994) and Quilici (1994) lack "cleanness" in the representation of the recognition problem. Specifically, the recognition algorithms are essentially a collection of heuristic tricks. This construction makes it difficult for one to perform a systematic analysis of different search methods and to understand how the addition of domain-specific knowledge affects performance. Wills (1992*b*) presents a detailed representation scheme (flow graphs for programs, and graph grammars for clichés) but does not directly exploit domain knowledge (plan library relationships) to constrain search for successful parses (explanations) of the source code.

A *Constraint Satisfaction Problem*[2](CSP) 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 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 problem definition that no two adjoining countries may be colored the same is represented as a constraint 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 possible colorings).

In this work, we present a generalized representation of program understanding as a CSP. For a given legacy source code, certain program components (explained later) are variables in the CSP. The domain values are the known program plans that may *explain* the task or purpose of each component. The CSP constraints are either *knowledge constraints* which describe how program plans may fit together to form larger plans, or *structural constraints*

[2]See (Kumar 1992) for an accessible and detailed treatment of this problem representation paradigm.

which describe how program components are structurally related, including the semantic relationships of Kozaczynksi et al. These constraints are exploited during integration of cliché instance integration to reduce the remaining search space. We refer to the program understanding CSP as PU-CSP, and define it in more detail in Section 3.1 on page 25.

(b) *Representing program templates recognition*

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. This mapping CSP, or MAP-CSP, is defined in detail in Section 3.1.3. 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.

This mapping or *recognition* procedure corresponds to finding one or all consistent matchings of the program plan in the source code. The PU-CSP representation extends the traditional CSP formulation by incorporating knowledge in the form of a hierarchy of program plans represented in terms of "Consist-of" and "Is-a" relations. This graphical representation may be seen as an AND/OR structure where the "AND" describes the union of sub-concepts in a concept, and the "OR" describes the set of possible specializations of a particular abstract concept. To exploit this extension we intend to design, implement and test search algorithms that can efficiently traverse this space and find a consistent explanation of the source program in terms of the hierarchy.

A primary advantage of this approach is its generality. By casting program understanding as CSP, the previously known search algorithms from that research area may be adapted. We also intend to perform a systematic study of different search heuristics, including both top-down and bottom-up as well as other hybrids, in order to discover their applicability.

2. *Efficiency*

   Most of the plan-recognition work has failed to focus on the heuristic adequacy in solving large problems. Quilici presents no empirical results whatsoever, and Kozaczynksi et al merely alluded to experiments with large source examples, and observe that their experiments resulted in excessive response times for concept matching as well as significant confusion with expert users with regards to concept representational language. No concrete examples or experiments are presented that might suggest that this approach has the potential to be used effectively in real situations. Wills presents empirical results that indicate the potential usefulness of a graph-based representation scheme. It is our belief that explicit application of knowledge constraints during a unified bottom-up and top-down recognition approach will improve upon this the graph-parsing recognition method in which concept components are effectively *decoupled* in grammar rules.

   Our primary focus is demonstration with large legacy code examples that an effective approach to program understanding is possible. 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.

1.5 Thesis completion conditions

The following conditions we have identified as sufficient to signal the completion of this research, culminating in the submission of a thesis dissertation:

- The definition of a model of program concepts or templates[3] in which the templates are represented as a special type of constraint graph. This constraint graph features a set of program template components constrained by relations such as call/calling, control flow, and data flow. A mapping between this constraint graph and some subportion of a piece of legacy source code indicates a potential instance of the program template in the source. The problem of identifying these matches is a constraint satisfaction problem referred to in this paper as MAP-CSP.

[3]Program templates have been similarly referred to in related literature as *abstract concepts* and *clichés*.

- The detailed description of the problem of understanding a given legacy source code as a constraint satisfaction problem (PU-CSP). This representation should provide for the interpretation of a program plan library as specific constraints in the problem, and include the heuristic approaches of earlier work in both plan recognition and software reverse engineering. Specifically, this approach must incorporate and utilize the conception of program template recognition (MAP-CSP) as part of an overall approach to understanding.

- The successful prototype implementation and demonstration of the heuristic adequacy of this new approach to program understanding. The determination of factors and conditions that inhibit heuristic adequacy at certain parameter settings such as legacy source code size will be an important result, particularly where the cost of obtaining additional "important" information can be determined. Explanation of the role of such an implementation as (1) an interactive tool for assisting experts in reverse engineering, and (2) a batch tool for simplification of existing legacy code through partial mapping to (and possible code replacement by) existing program libraries.

- The presentation of a set of test results that clearly place the effectiveness of this model within the scope of other published tools with similar goals. The only current known empirical results for an implemented model have been presented by Wills. In the case of sparsity of other similar experimental results, ours may be taken as one of the initial benchmarks in this area.

Proposal outline

We have presented an introduction of the main parts of our research in this chapter. The remainder of this paper is formatted as follows. First we summarize our proposal in terms of research goals and objectives, and provide task-level detail or milestones in the form of a timeline and dependency chart. Second, we present our current research within the context of initial feasibility results. Finally, in an appendix we list the primary references on which this work is based.

Chapter 2

Proposal : Research goals and objectives

2.1 Research proposal overview

We intend to describe, implement and practically demonstrate a new model of software program understanding extending previous work in both software reverse engineering and artificial intelligence plan recognition. Figure 2.1 presents a hierarchical breakdown of the components of this work.

There are four main areas of research interest within the larger task of specifying and justifying a new program understanding model:

1. *CSP Solver* in which a toolset of traditional tools for solving problems represented as CSPs is created. This toolset must be flexible enough to allow for the flexible *hybridization* of search strategies and constraint propagation methods. This toolset will support both the template matching methodology and the solution of the plan recognition CSP which will exploit domain knowledge (plan library) in controlling the solver tools. The two primary subtasks of this step are:

    - CSPs are typically solved with some kind of **Search Algorithm** such as BackTracking, ForwardChecking, BackMarking, or BackJumping.
    - **Constraint Propagation** methods can be applied to reduce a problem space *locally* in particular areas of the problem space, or

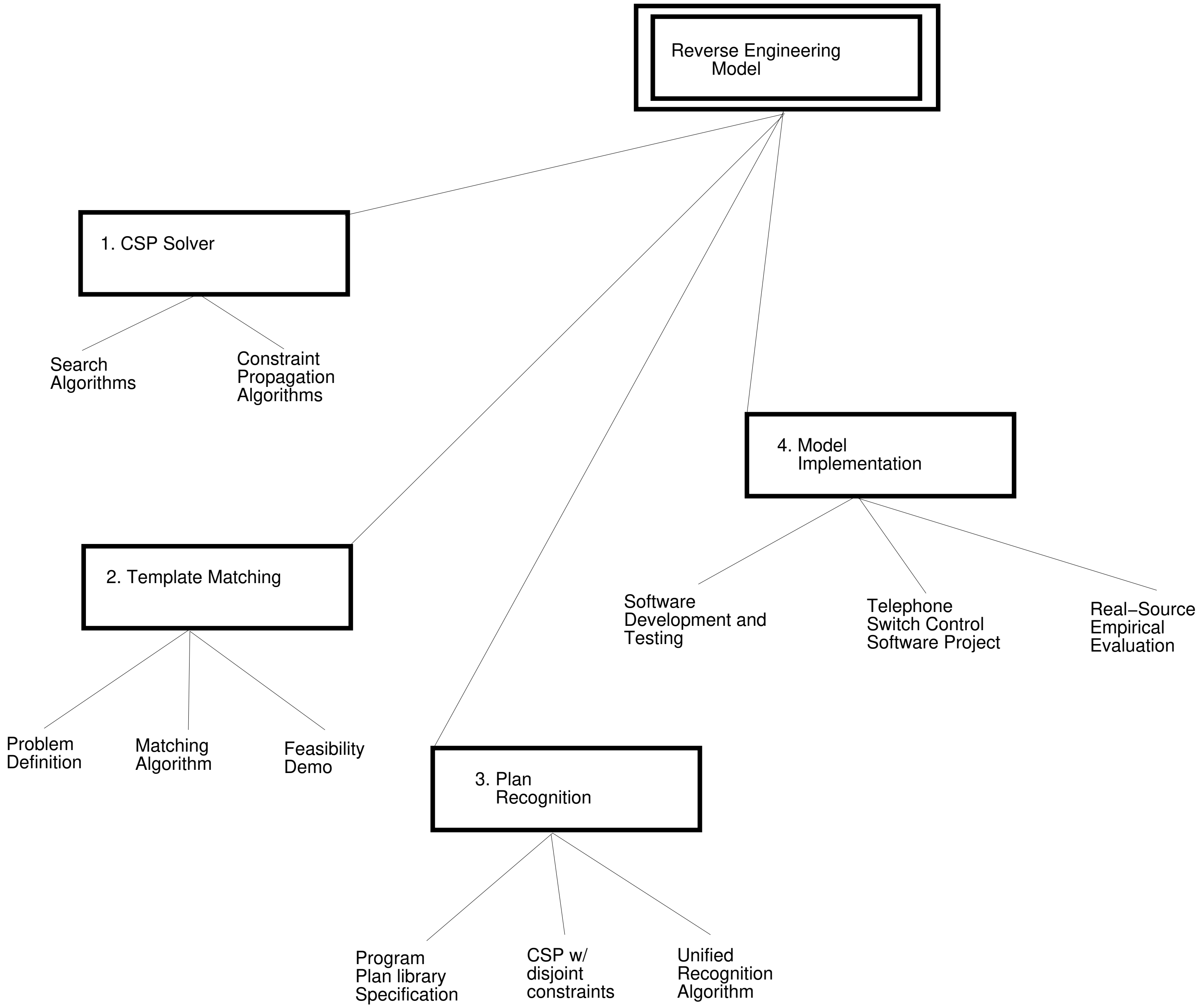

Figure 2.1: Research area overview chart.

globally across the entire problem. More intelligent search strategies may be seen as hybrids in which a simple search strategy is interleaved with some degree of constraint propagation at particular points in the search space.

2. *Template Matching*, in which a given programming plan template is matched against existing legacy source code. The primary parts of this task area are:

   - The **problem definition** encompassing the aspects of representation of programming templates in terms of components which may be matched to legacy code.
   - The template **matching algorithm**, defined in terms of a constraint satisfaction problem. This algorithm is essentially a traditional search algorithm with several domain-dependent heuristics utilized. Our earlier work with CSP representation and search algorithms in the spatial template recognition provided a basis for our extension of those approaches to the source code domain.
   - **Feasibility demonstration** of this algorithm is necessary in order to demonstrate the practical performance aspect of matching legacy code templates as part of the process of program understanding or reverse engineering. Since we are interested in constructing interactive tools we need to demonstrate real examples in near-real time. Spatial templates (Woods 1993*b*, Woods 1993*a*) were recognizable in near-real time for cases too *stretched* for human experts, and we have shown (Woods & Yang 1995*a*, Woods & Yang 1995*b*) a similar result for code templates in complex example legacy code. Kozaczynski & Ning (1994) recognize abstract program concepts in time ranging between seconds and minutes, perhaps applicable to batch processing but not interactive response.

3. *Plan Recognition*, in which a library of programming plan templates is searched in order to identify a mapping between existing legacy source code and possible programming plans. The complex interrelationship between a template matching paradigm, a general top-down and bottom-up understanding approach, and the particular domain

knowledge embodied in the plan library must be fully specified and justified.

- Any specification for the **program plan library** must be expressive enough to capture the expert knowledge which a program translator utilizes to interpret an existing source program *within a particular domain*. Specifically, knowledge about the typical program plans and their particular realizations must be supported, as well as additional relational information (such as indices and specializations) which may then be utilized to control and/or reduce search. A library must be constructed for the domain(s) of experimentation, and since these examples are drawn from real industrial legacy code, this library will be of non-trivial size and complexity.
- The program plan library is a representation that may be viewed as a set of constraints on possible instantiations of local explanations or domain values as part of the larger understanding process. These constraints are structured in a hierarchical format or AND/OR format. It is necessary to **extend the standard CSP representation** and search algorithms to account for this formulation.
- The **unified recognition algorithm** should encompass earlier work in plan recognition from the AI community which both attempts to reduce the range of search by adopting high-level plan hypotheses and fitting subsequent observations within this framework (top-down), and also to infer high-level plans based on instances of low-level actions and their interactions (bottom-up). The integration of the template matching algorithm into the overall understanding approach must be clearly specified in terms of both impact on the model structure and search performance.

4. *Model implementation*, in which our novel approach to program understanding as part of reverse engineering is implemented and demonstrated on sets of real source code provided through affiliation with the **telephone switch control software project**. Ultimately a computational algorithm or method is interesting for one or more of two

reasons: *functional (empirical) utility* in which a program may actually be used for a particular task, or demonstrate that a particular task may be accomplished in a particular amount of time, or *theoretical result*, in which a particular task is shown to be accomplishable (or not) with an algorithm of some complexity. We are concerned with demonstrating the functional utility of a limited and well specified method of program understanding as part of the larger problem of legacy code translation. Consequently, automatic understanding methods may be used as part of a partial-translation software, perhaps simplifying or reducing the size of legacy code. In addition, these methods may be used as part of an *interactive* tool assisting a human translator expert.

- **Software development and testing** of the matching and understanding algorithms is a necessary component of an empirical demonstration of an algorithm's functional usefulness. In particular, these algorithms will utilize and extend the CSP solver tools in both general and domain specific aspects.
- Our involvement in the **telephone switch control software project** allows us the opportunity to obtain a large legacy source example and attempt to apply our understanding and matching algorithms together with a specially constructed library to demonstrate the feasibility of our approach. The size and complexity of this example base places this domain well into the "real" and away from the "toy".
- Our results will take the form of a study and **empirical evaluation** of our implemented model's performance in terms of both partial automated translation of *actual* legacy code and interactive application with experts.

2.2 Research prioritization and timeline

Figure 2.2 presents a flow chart of the work outlined in this proposal, focusing on each of the four primary areas of interest previously outlined. Each task within an area is accompanied by an expected completion date, with current completed tasks indicated by a completion date of **December, 1994**. Figure 2.3 summarizes this information into a simple chart organized in terms of the four major checkpoints we have identified in the course of this research. We briefly outline each of the major subtasks of each research area **to be referenced as required**.

Area 1. CSP solver

1.1 Initial software development and testing

In earlier work (Woods 1993*b*, Woods 1993*a*) a constraint solving system based on search and constraint propagation was created which encompassed many of the traditional intelligent search algorithms and constraint propagation methods. A similar system has been created in LISP including BackTracking, ForwardChecking, BackJumping, and BackMarking search algorithms along with support for application of several typical domain-independent heuristics such as Dynamic Rearrangement of variables. The constraint propagation algorithm supported initially is the "standard" AC-3 algorithm.

1.2 Software extension and testing

Additional search algorithms and heuristics will be explored in order to develop sufficient comparison with other related work in both program understanding and plan recognition. These algorithms will be added to the initial development as they are needed, while the initial structure of the solver will remain intact. Similarly, additional CSP algorithms will be added as needed. In particular, any heuristic extensions which derive from the unique structure of the plan library will have to be accommodated.

Area 2. Template matching

2.1 Problem definition

The initial specification of code templates as a constraint satisfaction problem (MAP-CSP) was presented in (Woods & Yang 1995*a*, Woods & Yang 1995*b*) and is discussed in Section 3.1.3. This specification adapted the spatial template model presented in (Woods 1993*b*, Woods 1993*a*) to accommodate a subset of program constraint and attribute types.

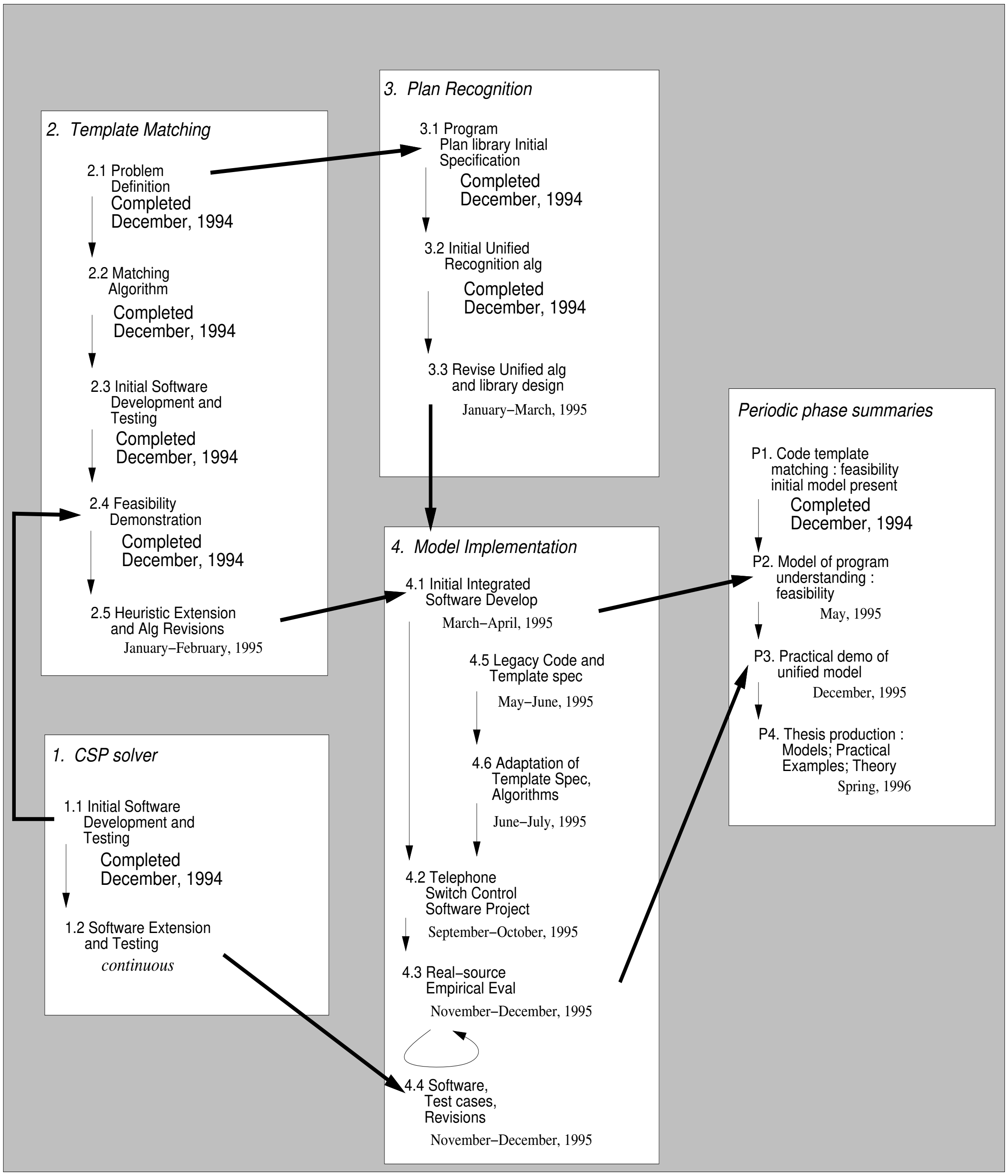

Figure 2.2: Research timeline chart.

Date	Major Milestones
Dec 94 **Completed**	1.1 CSP Solver: intial software develop 2.1−2.4 Template Matching: feasibility demo 3.1−3.2 Plan Recognition: initial recog alg Phase Write−up: P1 initial model feasible
May 95	2.5 Template Matching: extend, revise 3.3 Plan Recognition: revise model, lib 4.1 Implementation: integrated model Phase Write−up: P2 integrated feasibility
Dec 95	4.5−4.6 Implementation: legacy code adapt 4.2 Implementation: switch ctrl project 4.3−4.4 Implementation: evaluation, revise Phase Write−up: P3 practical model demo
Spring 96	4.4 Implementation: revisions, re−eval Phase Write−up: P4 Thesis production

Figure 2.3: Milestone chart.

2.2 Matching algorithm

Once specified, MAP-CSP is solvable via any combination of typical search and propagation techniques provided by the existing CSP-solver software as outlined in Section 3.2.

2.3 Initial software development and testing

Once specified, MAP-CSP was implemented in LISP according to the standard anticipated by the existing CSP-solver software.

2.4 Feasibility demonstration

It was observed in (Woods 1993*b*, Woods 1993*a*) that a non-trivial matching problem in the spatial domain could be solved using adapted search and propagation techniques. Further, the solution time was restricted enough that near-real time applications could be foreseen. In adapting the code matching problem as a CSP in the same way it needed to be shown that the new problem shared the desirable property of solution-in-reasonable time before it could be considered as an integral part of the larger PU-CSP. In Section 3.3 this result is demonstrated for non-trivial legacy code examples.

2.5 Heuristic extension and algorithm revisions

In the earlier work with spatial templates it was discovered that certain properties specific to the domain (spatial indexing and spatial locality) could be exploited to dramatically reduce the cost of applying certain common constraints, *and* also be exploited as a heuristic approach to domain reduction. In a similar vein, MAP-CSP is 1-dimensionally spatial, and could benefit at least partially from similar heuristics. We are currently investigating the possibility that other problem dependent heuristics could be exploited to narrow domain search.

Area 3. Plan recognition

3.1 Program plan library initial specification

In Section 3.1.3 the relation of program plans in the plan library is outlined. This library structure is adopted from Quilici (1994) and may be interpreted in a CSP as constraints on variable instantiation as described in Section 3.1.3's discussion of *knowledge* constraints.

3.2 Initial unified recognition algorithm

In Section 3.1 we specify the program understanding problem as a constraint satisfaction problem. Top-down reasoning is interpreted as both a particular form of constraint propagation during search and the selection and instantiation of variables during search. Bottom-up reasoning is represented through the integration of variable instantiations identified via template matching (MAP-CSP) into the search space.

3.3 Revise unified recognition algorithm, library design

In Quilici's plan library model additional knowledge such as index and specialization tests are represented and this information needs to be incorporated in the PU-CSP model, perhaps providing example-specific strict constraints, or perhaps only serving as preferential constraints.

While we currently have an algorithm for PU-CSP, the finer details of control are currently absent. In particular, questions of trade-offs between top-down and bottom-up approaches during search need to be determined by combining early empirical results with the intuitions obtained from the literature on previous plan recognition algorithms. A primary research task will be the adaptation of the algorithm to deal adequately with the hierarchical representation scheme by allowing for the hierarchical decomposition of legacy code and subsequent mapping of the various abstracted blocks to the hierarchical program library within the context of a single algorithm. This work will build on earlier work on Hierarchical CSPs (Mackworth, Mulder & Havens 1985).

Area 4. Model Implementation

While much of the implementation is done as-required, we indicate the tasks here which should have a definite completion date.

4.1 Initial integrated software development and testing

Two primary algorithms need to be implemented in order to arrive at a functioning understanding system: MAP-CSP and PU-CSP. These algorithms implicitly include all representation decisions regarding the knowledge of the program library. This task involves obtaining a *basepoint* functioning recognition system which interacts with legacy code and program plan library. Further extensions and revisions will still occur.

4.2 Telephone switch control project investigation, coding

In this work we are affiliated with Dr. Grant Weddell (Weddell 1994) in investigating the management of real large-scale software products at Bell Northern Research (BNR). One example of such a product is the software which controls a typical public switch such as a DS100 marketed by Northern Telecom. This software is approximately 250 megabytes, of which approximately 30 million lines is written in various (proprietary) source languages. Through our affiliation we will receive permission to access to large blocks of this source code, and also to abstracted concept representations of portions of this code. In order to test our approach on this software we must first construct a program plan library appropriate to this domain, and this library will be based at least partially on the existing abstract representations we receive. It is interesting to note that over 1000 programmers currently are employed at maintaining this code, and that even a small percentage decrease in the size and/or complexity of this code while maintaining functionality could have a dramatic affect on maintenance costs into the millions of dollars per year.

4.3 Real-source empirical evaluation

A primary result of our application of the aforementioned concepts and approaches should be a tool which may be used to map between source representations according to the partial automatic understanding of the source. This mapping can be exploited to suggest alternative representations for code portions, and hopefully actually replace repetitive code concepts with multiple applications of a single concept implementation, perhaps in the form of calls to a library of concepts representing existing software objects.

The overall effectiveness of our approach can be measured (1) in terms of usefulness as an interactive tool for an expert engaged in translation between source languages or in optimization or maintenance of software, or (2) in terms of effect as an off-line tool for automatically mapping between existing software and a library representation of existing software libraries as part of the process of simplifying existing legacy code by using existing new object libraries.

4.4 Software and empirical revisions

During the course of experimentation it will be necessary to incorporate certain new features and functionality to our existing understanding model.

4.5 Legacy code and legacy template specification

We will obtain the source language specification, and the actual source itself from BNR. We shall be meeting with BNR Feb. 20 1995 to discuss this and other related issues.

4.6 Adaptation of template specification and algorithms

With the arrival of the large, complex legacy code it will be necessary to adapt our template specification in the MAP-CSP and PU-CSP algorithms. Essentially while the initial implementation makes use of a simplified hypothetical language we must accommodate a richer language to encompass the more complex real domain example.

Periodic phase summaries

We have identified four primary checkpoints for document production during the course of this research. We describe them next.

P1. Initial model, template matching feasibility

An initial paper has been written which describes the overall approach in terms of extending earlier program understanding models by formalizing their representation within the CSP mathematical model, and by empirically demonstrating the applicability of one class of concept recognition as a part of this larger model.

P2. Model of program understanding, feasibility

Upon completion of an integrated model of program understanding, we will describe in greater detail the model, and present empirical results which support the use of this model. It is intended that this work be suitable for a major conference in software engineering or reverse engineering.

P3. Practical demonstration of unified model

Subsequent to application of our model to real legacy source obtained from industry, we will summarize the effectiveness of our approach to large real-world software. It is intended that this work be detailed and mature enough to submit to a software engineering journal.

P4. Thesis production

A detailed presentation of our model and in-depth analysis of the results of applying this model in real situations, especially in contrast to other known models.

Chapter 3

Feasibility discussion and model overview

> Versions of the material presented in this chapter have been submitted jointly with Qiang Yang for review to the *1995 International Joint Conference on Artificial Intelligence*, and to the *Seventh International Workshop on Computer-Aided Software Engineering.*

In this chapter we review the work currently completed (as described in the previous chapter) in terms of an overview of the entire problem of creating a model of program understanding suitable for matching source code templates to legacy code examples.

The first section of this chapter describes our representation of the program understanding problem as a constraint satisfaction problem (PU-CSP). Second, we introduce the range of search algorithms useful for the solution of CSPs in general and this CSP in particular, and position earlier work in program understanding within this context. Specifically we introduce the understanding problem as including the source code template instance identification task (MAP-CSP) as a subpart of the solution of PU-CSP. Finally, we describe our preliminary results which serve to demonstrate the feasibility of efficiently matching template instances (solving MAP-CSP) in reasonable sized legacy code examples.

3.1 Program Understanding as CSP : PU-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.

3.1.1 The Program Understanding Problem

Consider the C program outlined in Figure 3.1. 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.

```
main()
{
  char* A;   char* B;   char* C;
  A = "string 1";
  B = "string 2";
  C = "string 3";
  ...
  for (int i=0; B[i]; i++)
     print("%s",B[i]);
  ...
  for (int j=0; C[j];j++)
       print("%s",C[i]);
  ...
  for (int k=0;A[k]; k++)
       print("%s",A[k]);
  ...
}
```

Figure 3.1: Example C program.

```
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 3.2: Example abstract data type.

To understand this program, one might exploit a library of program plans which represents previously compiled knowledge about program composition within a particular domain. Figure 3.2 shows a program plan for the Abstract Data Type (ADT) or class **String** which is part of a library of plans. An excerpt of this library is shown in Figure 3.4. The ADT of Figure 3.2 provides a simple extension for manipulating character arrays which includes a buffer for storing the character array or string, and functionality for initializing and printing the string.

Given the source code in Figure 3.1, we would like to *understand* or *explain* some portions of the source program within known context of the program plan as represented in the **String** ADT. Succeeding at this could result in the replacement of much redundant source code with a single inclusion of the ADT. Figure 3.3 translation of the sample C program into C++ code, and highlights one instance of the String ADT recognized in the C source program.

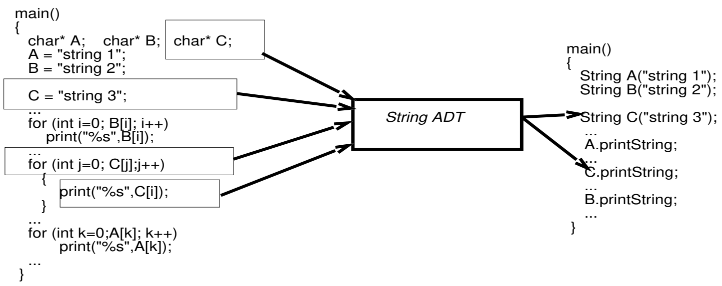

Figure 3.3: Desired C++ replacement of C code.

There are two advantages of identifying 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 mapping can be identified efficiently , and we outline how this mapping process may be integrated into the larger task of program understanding.

3.1.2 Quilici's approach

In Quilici's approach (Quilici 1994), program plans such as ADTs are represented as part of a hierarchical library of program plans as shown in Figure 3.4. Legacy source code in the form of an abstract syntax tree is mapped to the plan library through the use of indexes, 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 3.4 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 3.4, 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.

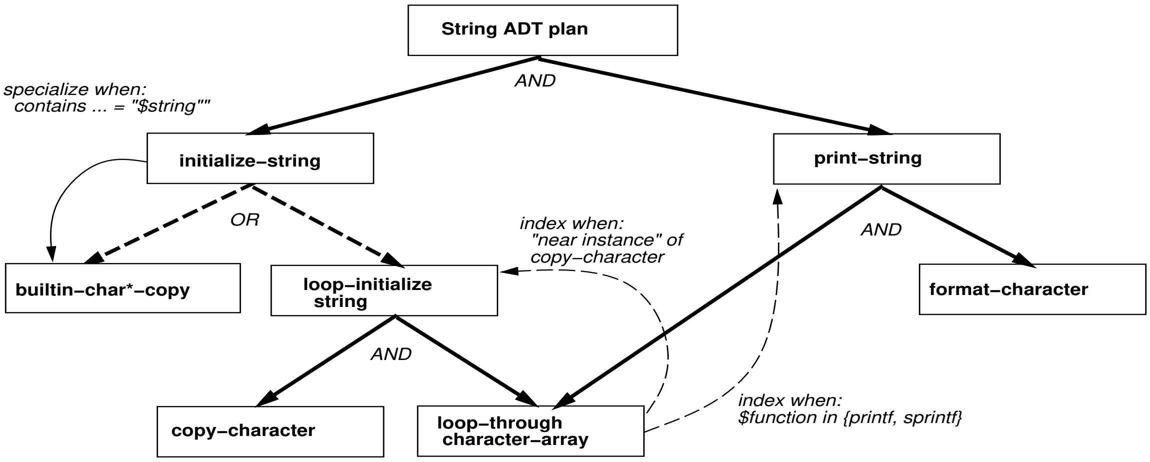

Figure 3.4: **String** ADT within a hierarchical program plan library.

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. A search is called *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, the search is *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.

Quilici's work is one of few which we have have identified that attempt to unify bottom-up and top-down program understanding paradigms within the context of a hierarchical library of program plans. There are, however, a number of shortcomings in his approach. First, there is a lack of a general mathematical model in his technique. Without this model it is not clear how one exactly should coordinate the top-down search with the 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 will scale-up when the problem size increases.

While reviewing 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 largely ignored 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 yet elegant mathematical model known as *constraint satisfaction*.

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.

3.1.3 Program template matching as CSP: MAP-CSP

A 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 3.5 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 "program statement type" (including instances such as **print** or

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.

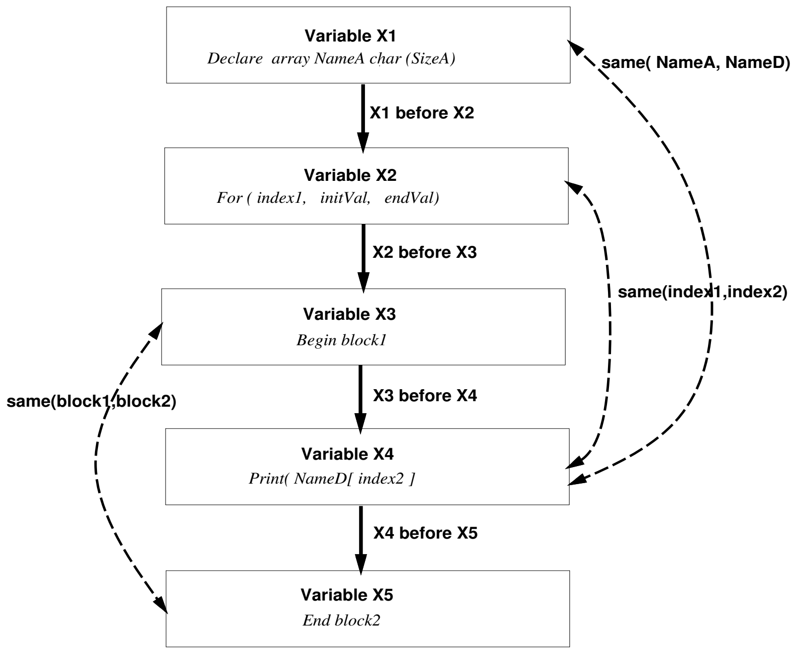

Figure 3.5: The **String** ADT template MAP-CSP.

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 shown in Figure 3.2. In one simple representation of this ADT as a plan template in a MAP-CSP shown in Figure 3.5, the variables are: $X_i, i = 1, \ldots, 5$. Initially the domain for each variable ranges through all source statements in Figure 3.1. The unary constraints in this example refer to program element (statement) type information, while other constraints relate to spatial ordering and data sharing. 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.

Program Understanding as CSP: PU-CSP

MAP-CSP is an integral part of the more ambitious task of understanding the complete legacy source program. We view this latter problem as the integration of local explanations of source code blocks. Each block of source code is first locally matched to a set of explanations in the plan library. Our task is to compose an explanation of the source code by selecting and piecing together sets of plan templates that are globally consistent according to the knowledge in the plan library.

As outlined earlier, our strategy of solving the program understanding problem is to model it as a constraint satisfaction problem. More precisely, a PU-CSP is formed in the following way. Suppose that an initial decomposition of the source code has been done by human or another system. Then 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 3.1 as the blocks. We take each block as a PU-CSP variable which ranges over all possible program plans of corresponding statement type, such declaration, assignment, print, etc, in the plan library of Figure 3.4.

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

(1) *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 Figure 3.1, 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.

(2) *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 3.1 were instantiated to program plan **builtin-char*-copy** as shown in Figure 3.4, then it is consistent to assign the last for-loop an explanation of **print-string** for the same string.

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. Additional examples and discussion of the ramifications of this interpretation of knowledge as constraints may be found in Woods (1994). In particular, this work describes in more detail the interrelation between search space expansion and knowledge constraint application.

Representing program understanding as PU-CSP provides a natural viewpoint for understanding Quilici's index tests as search strategies which are 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 the next section.

3.2 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 through heuristic backtracking algorithms which repeatedly select variables and instantiations and reduce remaining variable domains according to existing constraints. A thorough examination of these techniques is presented in (Nadel 1989, Kumar 1992). During a backtracking search, each variable instantiation is interpreted as extending the current understanding of a legacy program one step further.

In searching for one or one "good" solution, it may be worthwhile giving up "completeness" in order to dramatically speed search. One such single solution strategy that has shown great promise in various domains is referred to as "local search". Local search works by performing some random or informed initial assignment of domain values to each of the variables in the CSP, and then by attempting to repair the assignment according to some heuristic function. One common repair heuristic is to minimize the conflicts that arise with a particular assignment using a hill climbing approach. Local search has been shown in (Minton, Johnston, Philips & Laird 1992, Sosic & Gu 1990) to be orders of magnitude faster in finding single solutions quickly for difficult problems involving large variable and domain sets with many constraints. Recent ongoing work in the university course scheduling domain has also been discussed with similar results (Yang & Fong 1992). Local search can be adapted to find alternative solutions, however systematic local search strategies that would find all solutions eventually (ie local but complete) have not appeared to the best of this researcher's knowledge.

Some form of interactive search utilizing an algorithm for *all* solutions may be appropriate if it can be stopped after a "satisfactory" solution has been detected. In such cases where a complete strategy is used, the order of arrival at various solutions can be accommodated directly into the search strategy in the form of control heuristics for the selection of variables to instantiate, the order of domain values to attempt to assign, and for the control of expanding the search strategy deeper or wider in our quest for a "good enough" solution. One interactive adaptation of such an algorithm appears in (Woods 1993*b*).

Using the CSP representation, we can also consider a more systematic study of different search algorithms. Below is 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.

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 domain values for X_i;
2. [**Initial Constraint Propagation**] Reduce $Dom(X)$.
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 $Dom(X)$ is empty.
8. [**Solution Evaluation**] If V is empty, exit with Solution
(if all-solution, continue); else, goto Step 4.

The MAP-CSP specializes the generic algorithm with interpretation of variables as program template components or slots, domain values as potential legacy code components or blocks, and constraints as the template slot relations. Steps 4, 5 and 6 frequently interact tightly in domain-specific form in that variable and value selection and domain reduction may be achieved in tandem by certain heuristics such as limiting the search according to spatial or other properties of a particular template(Woods 1995, Woods 1993*b*, Lapointe & Prouix 1994).

In PU-CSP, the generic algorithm is interpreted so that variables map to program components, domain values map to potential program plans in the library, and constraints are both derived from legacy code relationships and domain constraints on plan interaction from the library. For the purposes of this work we assume the existence of an adequate "blocking" or "slicing" algorithms yielding either simple structural functions derived from the program's abstract syntax tree supplemented with semantic information such as call/calling, control/flow, and data flow relationships(Kozaczynski & Ning 1992, Quilici 1994, Wills 1990). More complex approaches that attempt to derive "related code portions" based on additional information such as dynamic program flow (execution) traces, other sophisticated analysis such as similarity analysis(Schwanke & Hanson 1994), or relatedness measures in problem decomposition (Yang 1995) may also prove useful in problem constraint. While variable determination is made according to some "blocking" algorithm, we observe that this process should not be one-dimensional, but rather hierarchical much as a program plan library is structured. In this way one may envisage "explaining" several code blocks through mappings and then the inferred "higher level" explanation maps to a larger code block consisting of a grouping of the smaller code blocks. While an abstract syntax tree typically describes contiguous code pieces, a more semantic decomposition might yield non-contiguous code blocks relationships.

Calls to MAP-CSP which attempt to identify or "scan" for mappings between sets of variables to sets of domain values circumvent the variable-by-variable process of the generic CSP solution strategy. This call may be made in-advance of the search process at Step 1 during initialization thus giving search a solid starting point from which to "focus" search. In addition, other algorithms may determine heuristically that a call to identify template instances may be profitable during search such as at Step 6. An algorithm supporting constraint propagation given a hierarchically structured domain value set is described in Mackworth et al. (1985).

The CSP solution algorithm contains several choice points which both individually and in combination affect the resulting search performance. These choice points are explained as follows:

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 $Dom(X)$ 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.

Constraint Propagation is the reduction of domain ranges locally or globally within the CSP problem graph. Existing algorithms include AC-3(Mackworth 1977), AC-4(Mohr & Henderson 1986), AC-5(Van Hentenryck, Deville & Teng 1992), and other variations(Freuder 1982, Cooper 1989).

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.

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.

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 (Nadel 1989) the advantages of exploiting various algorithms for achieving a limited degree of partial consis-

tency amongst variable sets are examined.
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[1] 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.
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 solution for adequacy. Similarly, if there existed particular measures of adequacy or *soft, preferential* constraints that may have been relaxed during search, such a measure could be applied here.

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* (Mackworth et al. 1985). 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 by other source functions, and the knowledge constraints in the program plan library. In Woods (1994) a closer examination is given to controlling search using plan knowledge. If we are given two source components each "explained" through some mapping, and these components or plans are known to be part of a common plan structure, a determination of the *connection*, that is, the abstract plan that includes both plan instantiations is an *explanation* of the relationship of the two plans. Many plan recognition approaches imply that the "lowest" common parent explaining a set of instances is the sensible inference and should be adopted in the default case. This reasoning is ubiquitous in the literature, despite many variations. Charniak & McDermott (1985)[560-562] provides a clear example of this application of the "Occam razor principle" in the recognition domain. This process corresponds to the **bottom-up search**

[1]These and other intelligent backtracking algorithms are described in detail by Nadel in(Nadel 1989).

of Quilici's approach.

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, which can potentially reduce the number of domain sizes.

In the generic search algorithm a set of now familiar choice points in Quilici's work 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 (Haralick & Elliott 1980), 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, 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(Woods 1993*b*).

3.3 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 3.5 a CSP is described for the **String** ADT. This CSP contains 5 variables each corresponding to a program plan of the ADT in Figure 3.2. 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

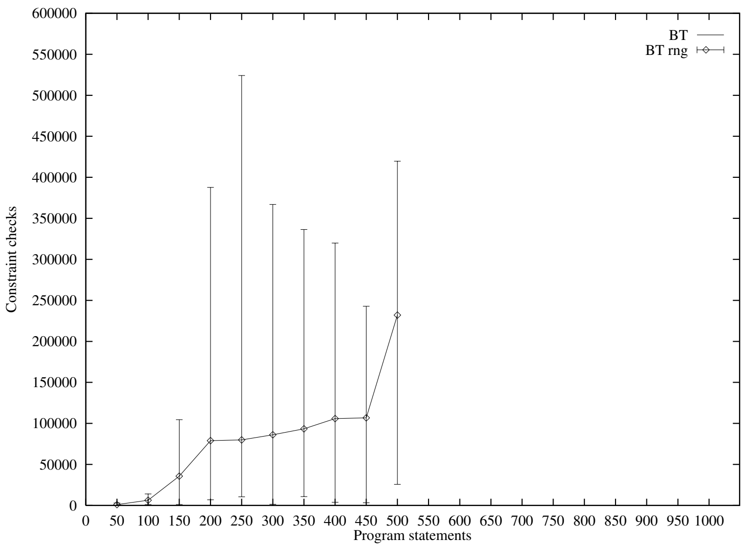

Figure 3.6: Matching using Standard BackTrack.

the program template in a sample source code, and by adding some variable amount of additional program statements as "noise" around the template instances [2]. In Figure 3.1 there are three separate instances of this ADT.

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 3.6, with a CPU time bound of 600 seconds, for *Standard BackTracking*; in Figure 3.7 for Forward Checking with Dynamic Rearrangement; in Figure 3.8 for AC-3 constraint propagation in advance of Standard BackTracking; and in Figure 3.9 for AC-3 in advance of Forward Checking. Each figure shows the number of constraint

[2]The experiments presented in this paper are undertaken using artificially generated.

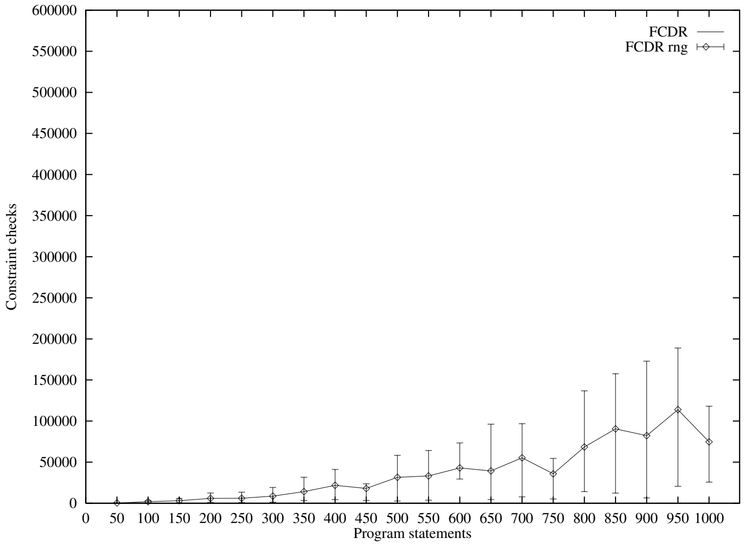

Figure 3.7: Matching using Forward Checking(DR).

checks performed with a 95% confidence interval. The number of constraint checks has been observed to be directly proportional to the amount of CPU time required for a solution, with 50000 constraints checked in roughly 100 CPU seconds.

We wish to demonstrate that the MAP-CSP representation and algorithm was capable of providing all-instance results. 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: First, *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.

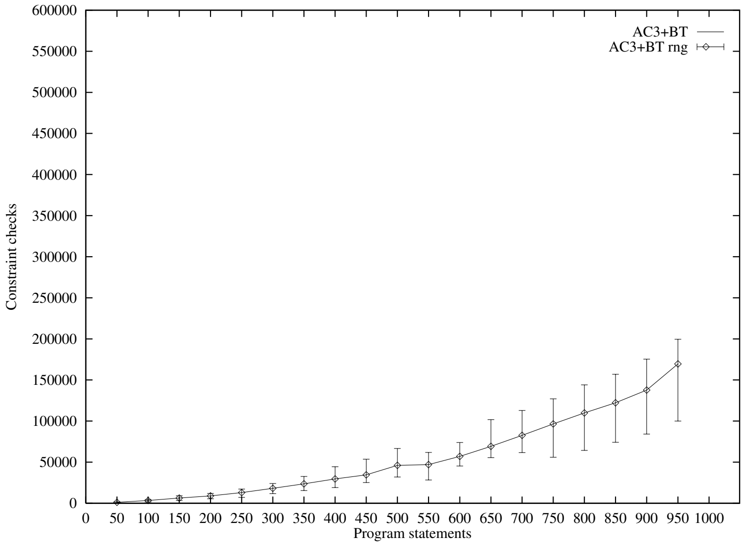

Figure 3.8: Matching using AC-3 and Standard BackTrack.

Second, 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 translating a legacy code.

Third, in experiments where the number of source lines exceeded 200, the appearance of *false* solutions 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.

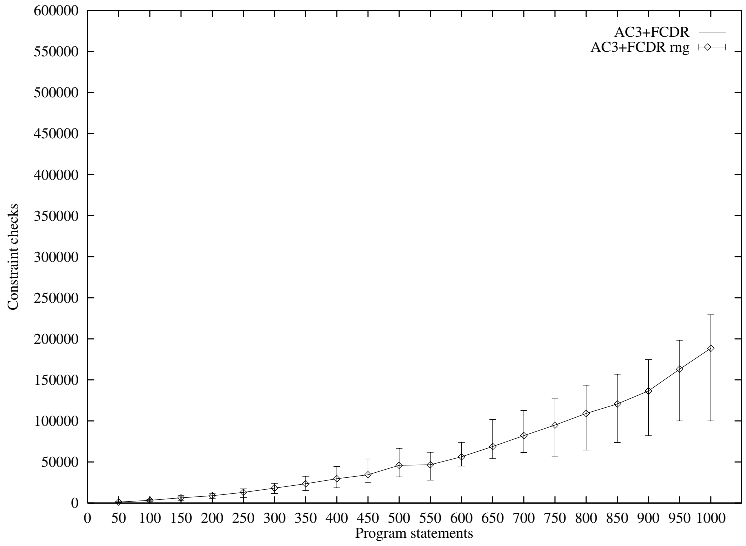

Figure 3.9: Matching using AC-3 and Forward Checking.

3.4 Feasibility conclusions

In this section we have specified a general representation of the program understanding task as a constraint satisfaction problem. Two primary parts of this 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 control strategies coupled with 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 described and implemented a portion of the PU-CSP as a template matching problem (also as a CSP), MAP-CSP. We 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 larger, possibly interactive, program understanding task.

Chapter 4

Conclusion

Recently there has been intense interest into the prospect of reverse-engineering existing software so as to obtain design information, generate documentation, or assist in re-engineering functions such as maintenance and extension. The expert charged with understanding a large block of legacy code written in a possibly unfamiliar language and domain faces a daunting task.

Past work in plan recognition has sought to represent human domain knowledge as hierarchies of plans describing actions and goals, and presented matching processes between input and plan hierarchy as methods of *understanding* input within some domain. This work has typically applied only to small "toy" domains. Recent advances in software reverse engineering(Quilici 1994, Kozaczynski & Ning 1994, Wills 1990, Rich & Waters 1990) have applied such matching methodologies to limited source code examples and demonstrated how these simple examples could be contextualized in a library of programming plans. Some of this work (Kozaczynski & Ning 1994, Wills 1990) has made use of the idea of recognizing abstract concepts as part of an overall program understanding approach. These approaches utilize search is various degrees, but fail to tie the representational schemes and subsequent represented knowledge to search control, thus severly limiting what may be predicted about algorithm performance given a particular set of knowledge. From this set of work in reverse engineering of program source, only Wills has attempted to address the issue of heuristic adequacy or presented well-defined example situations and empirical results. Unfortunately, Wills fails to explicitly describe the integration of program plan knowledge and representation, the heuristics described by Quilici and Kozaczynski et al, and abstract concept instances within a single framework of program understanding.

In this work we present a unified model of program understanding that connects "traditional plan recognition" approaches to knowledge representation and understanding methodologies and current software reverse engineering program understanding models. This connection is made via the representational scheme of constraint satisfaction, and results in two new CSP algorithms: one for program understanding, and one for abstract concept (template) recognition. This two-pronged model incorporates program plan library knowledge, legacy source code relational constraints, and earlier understanding and reverse engineering heuristics. We present not only a unified theory of program understanding for software reverse engineering, but demonstrate empirically the effectiveness of this theory when applied to real legacy code.

Primary references

Software reverse engineering

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

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

3. Wills, L. M. (1992*b*), Automated program recognition by Graph Parsing, PhD thesis, MIT. See http://www.cc.gatech.edu/cogsci/faculty/wills/phd-thesis.html.

4. Wills, L. M. (1990), 'Automated program recognition: A feasibility demonstration', *Artificial Intelligence* **45**(2), 113–172.

5. Rich, C. & Waters, R. (1990), *The programmer's apprentice*, Addison-Wesley, Reading, Mass.

Plan recognition

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

7. Mooney, D. J., Carberry, S. & McCoy, K. F. (1991), 'Capturing high-level structure of naturally occurring, extended explanations using bottom-up strategies', *Computational Intelligence* **7**, 334–356.

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

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

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

11. James F. Allen and C. Raymond Perrault. Analyzing intention in utterances. *Artificial Intelligence*, 15:143–178, 1980.

12. James F. Allen and C. Raymond Perrault. Participating in dialogues: Understanding via plan deduction. *CSCSL*, 1978.

Constraint satisfaction problems

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

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

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

Acknowledgements

Thanks to Qiang Yang for many helpful discussions and suggestions in the construction of the ideas presented in this paper, and for careful and experienced editing. Steven Woods acknowledges the support of the University of Waterloo Department of Computer Science, the Institute for Computer Research (ICR), the Information Technology Research Centre (ITRC), and the National Science and Engineering Research Council of Canada.

Bibliography

Allen, J. F. & Perrault, C. R. (1978), 'Participating in dialogues: Understanding via plan deduction', *CSCSL*.

Allen, J. F. & Perrault, C. R. (1980), 'Analyzing intention in utterances', *Artificial Intelligence* **15**, 143–178.

Carberry, S. (1985), Pragmatic Modeling in Information System Interfaces, Department of computer and information sciences technical report 86-07, University of Delaware.

Carberry, S. (1988), 'Modeling the user's plans and goals', *Computational Linguistics* **14**(3), 23–37.

Carberry, S. (1990), 'Incorporating default inferences into plan recognition', *Proceedings of the 8th AAAI* **1**, 471–478.

Charniak, E. & McDermott, D. (1985), *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company.

Cooper, M. C. (1989), 'An optimal k-consistency algorithm', *Artificial Intelligence* **41**, 89–95.

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

Grimson, W. E. L. (1990), 'The combinatorics of object recognition in cluttered environments using constrained search', *Artificial Intelligence* **44**, 121–165.

Haralick, R. & Elliott, G. (1980), 'Increasing tree-search efficiency for constraint satisfaction problmes', *Artificial Intelligence* **14**, 263–313.

Kautz, H. & Allen, J. (1986), Generalized plan recognition, *in* 'Proceedings of the Fifth National Conference on Artificial Intelligence', Philadelphia, Pennsylvania, pp. 32–37.

Kozaczynski, W. & Ning, J. Q. (1992), 'Program concept recognition and transformation', *Transactions on Software Engineering* **18**(12), 1065–1075.

Kozaczynski, W. & Ning, J. Q. (1994), 'Automated program understanding by concept recognition', *Automated Software Engineering* **1**, 61–78.

Kumar, V. (1992), 'Algorithms for constraint-satisfaction problems', *AI Magazine* pp. 32–44.

Lapointe, S. & Prouix, R. (1994), Fuzzy geometric relations to represent hierarchical spatial information, *in* 'Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence'. to appear.

Mackworth, A. (1977), 'Consistency in networks of relations', *Artificial Intelligence* **8**, 99–118.

Mackworth, A., Mulder, J. & Havens, W. (1985), 'Hierarchial arc consistency: exploiting structured domains in constraint satisfaction problems', *Computational Intelligence* **1**, 188–126.

Minton, S., Johnston, M., Philips, A. & Laird, P. (1992), 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems', *Artificial Intelligence* **58**, 161–205.

Mohr, R. & Henderson, T. (1986), 'Arc and path consistency revisited', *Artificial Intelligence* **28**, 225–233.

Mooney, D. J., Carberry, S. & McCoy, K. F. (1991), 'Capturing high-level structure of naturally occurring, extended explanations using bottom-up strategies', *Computational Intelligence* **7**, 334–356.

Nadel, B. A. (1989), 'Constraint satisfaction algorithms', *Computational Intelligence* **5**, 188–224.

Paul, S. & Prakash, A. (1993), Generating programming language-based pattern matchers, *in* 'Proceedings of CASCON '93', Vol. 1, IBM Canada/NRC Canada, pp. 227–243.

Quilici, A. (1994), 'A memory-based approach to recognizing programming plans', *Communications of the ACM* **37**(5), 84–93.

Rich, C. & Waters, R. (1990), *The programmer's apprentice*, Addison-Wesley, Reading, Mass.

Schwanke, R. W. & Hanson, S. J. (1994), 'Using neural networks to modularize software', *Machine Learning* **15**, 137–168.

Selfridge, P. G. (1994), 'Report on the First Working Conference on Reverse Engineering, May 21-23, 1993, Baltimore, Maryland, In conjunction with the International Conference on Software Engineering', *Automated Software Engineering* **1**, 61–78.

Sosic, R. & Gu, J. (1990), 'A polynomial time algorithm for the n-queens problem', *SIGART*.

van Beek, P., Cohen, R. & Schmidt, K. (1994), From plan critiquing to clarification dialogue for cooperative response generation, advance version of journal paper.

Van Hentenryck, P., Deville, Y. & Teng, C.-M. (1992), 'A generic arc-consistency algorithm and its specializations', *Artificial Intelligence* **57**, 291–321.

Weddell, G. (1994), Project proposal for office date management, Personal communication.

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

Wills, L. M. (1992*a*), Automated program recognition by Graph Parsing, AI Laborary Technical Report 1358, MIT.

Wills, L. M. (1992*b*), Automated program recognition by Graph Parsing, PhD thesis, MIT. WWW available as http://www.cc.gatech.edu/cogsci/faculty/wills/phd-thesis.html.

Woods, S. (1993*a*), Interactive recognition of spatially defined model deployment templates, Research memorandum, Combat Intelligence Automation Group, Defence Research Establishment Valcartier, Courcelette, Quebec CANADA. unclassified.

Woods, S. (1993*b*), A method of interactive recognition of spatially defined model deployment templates using abstraction, *in* H. Merklinger, M. Farooq, P. Roberge & R. Grodski, J.J.and Dobson, eds, 'Proceedings of the Knowledge -Based Systems and Robotics Workshop', Department of National Defence, Government of Canada, pp. 665–675.

Woods, S. (1994), Knowledge based model recognition in program understanding using constraint satisfaction, Cognitive Modeling termpaper submitted to Dr. Paul Thagard, University of Waterloo, Waterloo Ontario.

Woods, S. (1995), On the utility of explicit application of spatial locality features in the automatic recognition of spatial templates, in progress.

Woods, S. & Yang, Q. (1995*a*), A constraint-based approach to program plan recognition, submitted for publication.

Woods, S. & Yang, Q. (1995*b*), Program understanding as constraint satisfaction, submitted for publication.

Yang, Q. (1995), Decomposing problem spaces into non-interfering operator sets, Personal communication.

Yang, Q. & Fong, P. (1992), Solving partial constraint satisfaction problems using local search and abstraction, Technical Report cs-92-50, University of Waterloo.