Knowledge based model recognition in program understanding using constraint satisfaction

Steven G. Woods
Logic Programming and Artificial Intelligence Group,
Computer Science Department, University of Waterloo,
Waterloo, Ont. Canada, N2L 3G1.
email: sgwoods@logos.uwaterloo.ca

January 24, 1995

Abstract

Source code translation across programming paradigms is a difficult, experience-intensive task. One important part of this task is understanding the "source" program to be translated. Recent work in program understanding has focused on attempting to identify mappings between a known hierarchical library of program plans and the program source code. Much of program understanding research, like plan recognition research, attempts to incorporate knowledge about the way humans recognize plans in order to speed the mapping process. One heuristic observed during human textual comprehension is broad context-forming behaviour before actual consideration of the bulk of the source material. This "scanning" behaviour may be thought of as identification of features the *scanner* considers important. We attempt two primary tasks in this paper: (1) to represent the program understanding task in a framework that will allow a clear interpretation of the process of matching source to the knowlege hierarchy, and (2) to integrate into this representation some earlier work in spatial template recognition as a part of the identification of "important" features during the expert scanning phase of understanding.

Specifically, in our implementation and experiments, we represent as a constraint satisfaction problem the task of recognizing the task of utilizing the functionality and data storage features of a particular known abstract data type. This implementation is undertaken on top of the existing MOTORS (Woods 1993*a*) constraint satisfaction toolset, written in LISP. We run a set of tests across several simple but well understood CSP solution strategies and demonstrate the applicability of this recognition approach, even in noisy situations. Throughout this paper, we attempt to integrate how (simplified) representations and models of human cognition in program the understanding task may be integrated into our single representation of program understanding as a

constraint satisfaction problem. The actual implementation of program understanding as a CSP and encapsulation of known understanding models into this framework forms the larger body of our ongoing research.

1 Introduction

The problem of understanding existing source code is a critical task for any human expert undertaking to translate[1] source code from one language to another. The successful interpretation of an explicit representation of knowledge made by another agent is one key aspect of human intelligence.

Earlier (Allen & Perrault 1978, Allen & Perrault 1980, Kautz & Allen 1986, Carberry 1988, Carberry 1990) work in plan recognition as well as recent program understanding efforts outlined in (Quilici 1994) have attempted to capture some simplified intuitions of this type of human understanding or comprehension through modeling existing human domain-specific knowledge as hierarchies of plans that describe relevant actions and plans in that domain. Given a source input such as action observations for problems in the plan recognition domain, or source code in the case of program understanding, one needs to be able to construct a mapping from the "non-understood" *source* to the "understood" *target* knowledge hierarchy. In this way, recognized "small" plan portions may be contextualized in terms of the larger hierarchy through *bottom-up* reasoning which (1) "explains" the appearance of a low-level code part by determining which high-level program plans make use of that code part, and (2) attempts to find a common higher-level context to explain groups of lower-level actions or perceptions. With the recognition of some context in place, it is possible to guide search for, or to expect, other elements of the hierarchy according to the believed-context of the previous actions. This type of reasoning is known as *top-down.* A single recognized entity may have many possible contextualizations in a particular knowledge hierarchy, and simply trying to fit many of such recognitions into one or more allowable total contexts can be a very expensive process computationally. Similarly, top-down strategies which attempt to guide the possible contextualizations by adopting larger contexts earlier in the integration process can suffer from early over-commitment and result in the need to try alternative contexts after much work has been expended on an errant context commitment.

Some recent work in constructing models of program understanding (Quilici 1994) has sought to ameliorate these difficulties by combining top-down and bottom-up strategies according to some common intuitions about human reasoning with plans and knowledge. This work attempted to formalize the structure of plans within a plan hierarchy or library so as exploit additional heuristic knowledge that was observed in human experts. This additional

[1] Computer science literature dealing with assemblers and compilers such as (Calingaert 1988) sometimes refer to *translation* as the entire process of changing high level source code to machine executable instructions. For the purposes of this article, we interpret "translation" of programming languages at a higher level, specifically from one source language into another source language.

reasoning knowledge includes a preference to specialize general plans as much as possible, and a memory which is able to quickly associate a particular recognized plan with other plans that are commonly encountered in context with the base plan, provided some set of sufficient conditions have been met.

Quilici (Quilici 1994) presents a simple algorithm which is able to integrate individual recognitions into larger contexts using these heuristics. This approach can be viewed as a controlled search through all possible contextualizations of the individual low-level recognized entities. A state in the search space is essentially a set of partial-instantiations of source components and plan-hierarchy "explanations". A solution is a complete instantiation in which all entities are contextualized (explained), and all of the rules for plan-relation encoded in the plan-hierarchy hold. In many ways this past work is "messy" in that no description of the resulting search space has appeared, and consequently it is difficult to evaluate the interactions of the different heuristics and to interpret how additional control strategies might affect the performance of such an approach. A "neat" approach in contrast would offer an easily describable search space that can be analyzed to help determine what role additional domain knowledge may plan in reducing search, and consequently improving performance.

Some previous work represented the problem of recognizing spatial templates in complex, noisy situations as a constraint satisfaction problem (CSP)(Woods 1993*b*, Woods 1993*a*). In this work, a spatial model to be recognized is represented as a constraint graph in which features of the model are variables ranging over possible specific-situation instances, and spatial, functional, and structural relationships are modeled as constraints amongst the features. The advantage of such an approach is that it allows one to readily conceive of the problem in terms of efficient traversal and reduction of search spaces as well-discussed in the literature of both constraint satisfaction and search, and thus provides a basis in which to apply known constraint propagation methods, typical CSP search strategies and specific heuristic approaches to solving these problems. The search space resulting from a particular choice of control strategy and heuristics can be examined, and the order and manner in which constraints are utilized or applied can be critiqued with the intent of formulating more systematic and efficient approaches.

In this paper we model program understanding as constraint satisfaction, and the solution of this problem explicitly as CSP search. Bottom-up and top-down strategies are re-interpreted in CSP search as different types of domain reduction and variable instantiation. Additionally, where previous recognition approaches have been very non-focused in the selection of subgoals and areas of interest in problems, we explicitly model focus as variable and domain-value selection. The resulting system described in this work may be viewed as a knowledge-guided focus planner for constraint satisfaction problems.

The format of this paper is as follows. First we will briefly motivate our interest in program translation, and identify the two main parts of this task, program understanding and program construction. We outline one example of automated program understanding, and identify the difficulties with such an approach. Program understanding involves both *global* interpretation of a large source code, and *local* interpretation in which quick identification of "common" plans can greatly aid the larger understanding task. Second, we will provide a brief overview of problem representation and solution using constraint satisfaction, and detail the representation of the understanding task within the paradigm of CSP, representing hierarchical knowledge novelly in the form of constraints. Third, we present some experimental results which detail the effectiveness of local interpretation and the ability of local plan identification scales up over larger sources. We discuss how these results generalize across other related recognition problems. Finally, we outline the future direction of this work in progress, with emphasis on future extensions including broader implementations, larger plan hierarchies, and more *global* interpretation of source examples.

2 Program Translation

As corporate or academic goals and preferred paradigms change, a current problem faced by many programming institutions is the translation of massive amounts of existing source code into different source language(s). Frequently this existing source code has evolved over many years and has widely varying degrees of design or programming documentation available to aid the translation programmers. Direct or "rote" translation is usually not possible between source languages since different languages usually contain differing capability sets, and frequently embody entirely different approaches to programming. A recent example of one such significant paradigm shift in programming that is having a great impact on the computing community is the increased use of object oriented programming languages such as $C++$.

One major goal of object oriented programming in general, and $C++$in particular, is to make available *libraries* of programming functions that are well encapsulated, with very clear understanding of function in terms of a contractual-type agreement of input and output. Functionality is thus localized behind boundaries and type-safe declarations. Libraries of this sort allows a programmer to easily use provided objects for a variety of particular problems, thus avoiding the repetitious creation of similar code and allowing for various systems to share the advantages of only maintaining one code source for the same functionality. There are several major challenges to this type of code-sharing, ranging from simple knowledge of the existence of such code, even within an organization, to inadequate documentation of existing code, making it difficult for a programmer to identify the purpose or usage of the code. Currently, this translation is accomplished through the hands-on expertise of programmers who have a thorough **knowledge** of the *source language*, the *target language*, and usually the *domain of application* as well.

Program Understanding

It is understood that such an expert is well versed in common programming *structures* and *methods* for constructing source code. Typically, such an expert is capable of recognizing both general and domain specific programming plans, one general instance of which would be a loop structure. The expert is capable then of mapping this structure to another known structure or configuration in the target language that serves the same *purpose* or programming plan. In effect, the expert recognizes the original programmer's intent or programming plan and maps this to an instantiation in the target language. Thus, we divide translation into two subtasks: recognition of programming plans in the source language, and construction of programming plans in the target language. This paper focuses only on the first subtask, recognition or understanding of programming plans.

Expertise[2] in recognizing general programming plans is heavily dependent on past experience. In order to recognize programming plans, one needs to know both how these plans may be realized in the source language, and also what specific instances of these general realizations "looks" like. Essentially one needs to know about *how* code reflects programming plans. While these programming plans may be realized in many widely varying implementations, some plans are quite typical and can be recognized more easily. In addition, some programming plans are less common than others and subsequently recognizing a rare plan may serve as an "unique feature identifier" which allows a limited possible range of explanations for the remainder of the source. Since actual realizations of programming plans may varying widely, experts must be able to wade through program clutter or "noise" and recognize the salient features of known programming plans. Computer professionals are well aware of the difficulty of maintaing poorly or quickly written and inadequately documented code. This type of code is *difficult to understand*; in our vernacular, these programs are difficult to map to a knowledge base of programming plans. As demonstrated in a spatial pattern recognition domain in (Woods 1993*b*), situation clutter can overwhelm a human expert's ability to detect familiar model instances. Since noise plays such an important part in "confusing" the recognition of models, the existence of tools which are capable of quickly locating model instances, even in noisy situations, could be of great assistance to an expert as part of an interactive recognition or understanding system. These instances can be mapped to the program plan library giving initial contextual clues to an ongoing understanding task, and thus reducing the size and scope of the overall translation task with which the expert must contend.

The problem of recognizing program plans has previously (Quilici 1994) been presented as a problem in reverse engineering, where one wishes to determine the programmer design goals which were used in the construction of source code. There are at least three critical elements involved in this task:

- *Source Representation* The program instructions source, data, and control flow must be represented in some fashion.
- *Plan Library* Both domain-independent and domain-dependent programming plans must be captured in some form of library.

[2]The nature of this expertise is one focus of our ongoing research.

- *Source-Library Mapping* Mappings must be identified between the source representation and the programming plan library with the use of some specific algorithm. These mappings are *explanations* of parts of the source in terms of existing represented knowledge.

In order to locate mappings between source code and some specific plans existing in the plan library, we observe that both the representation of the source and the specification of plan templates in the source "language" are crucial. In order to actually perform this identification, some matching algorithm dependent on the representations is required, and in order to *understand* the source, the entire source code needs to be mapped to the knowledge contained in the programming plan library.

The source code is initially viewed as simple program text which, by definition, embodies one or more programming plans. The source consists of lines of parsed source, possibly in the form of an abstract syntax tree, where the connection as defined within the context of the domain between portions or functional blocks of the source code is not immediately discernible. The understanding process involves the determination of how the text coheres and maps to an explicit knowledge library of programming plans. Besides an abstract syntax tree constructed from units of code at a level including operators and parameters, additional information at a higher level may be extracted from the source, specifically, useful delineations of groups of code units and lines into functional blocks or procedures can be formulated. These blocks can be used as a convenient "higher" level of abstraction for explanation purposes. Each functional block potentially participates in relationships with other blocks, and information about relative position of these blocks, or possibly the type-nature of the interface between blocks may be useful in explaining the purpose behind the use of these blocks by a programmer.

Programming plans may be readily expressed hierarchically, and programmers typically make use of this hierarchical structure in program analysis, design and finally, construction. General plans such as **Loop-through-all-set-values** have many possible specific instantiations which realize the higher level plan, for example **Iteration-through-array**, or **Traverse-linked-list**. We may view our entire collection of programming plans as a library of this nature, with each plan potentially serving as a part of a higher level programming plan, and possessing several possible specializations. Figure 1 shows a simple plan library excerpt formed around code designed for matching spatial objects. The **Attribute-Match** plan necessarily specializes as either **Type-Compare**, *or* **Orientation-Match**, *or* **Location-Match**. Additionally, the **Type-Compare** plan is composed of a **Global-Star Match** plan *and* a **Type-Key Equality** plan. Figure 2 outlines some source code from this domain in which an instance of a **Global-Star Match** and a **Type-Key Equality** combine to form an instance of **Type-Compare**. The plan library represents knowledge about how programming plans may be realized. A complete mapping from this library to source code is an *explanation* of the source code insofar as the source may now be viewed as an instantiation of some portion of our knowledge about how programs are structured, and subsequently expressed in the target language. A partial mapping similarly *explains* the mapped portion of the source code.

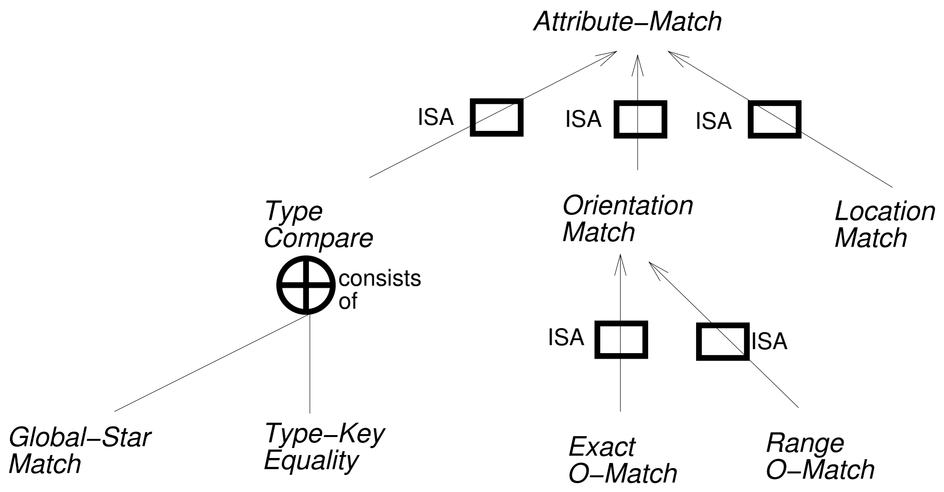

Figure 1: Plan library hierarchy excerpt.

We wish to reduce the size of the translation problem by automatically mapping some portion of the source code to target library structures. In addition to to reducing the size of the translation problem, partial mappings of the source code to target library structures can assist an expert in the process of *understanding* the remainder of the source code. Either the remaining source can be contextualized within the scope of our explicit programming plan library, or the library needs to be extended so as to accommodate "new" programming plan knowledge[3]. The partial mappings indicate some set of "active" portions of the plan library and serve as clues as to the higher level programming plans that are contained in the source. This information can suggest which additional programming plans may be present in the source, and consequently suggest future matching possibilities. Essentially the program understanding task can be viewed as a continual process of *recognition* of familiar or expected plans which explain local portions of the source, and *integration* of these recognized plans into a global context which identifies the overall programming plan evidenced by the whole source code.

We have described how the task of translation of source to target as consisting of two primary steps or phases: *program understanding* (or plan recognition) in which the intent is to determine the programmer design goals which were used in the construction of the source code, and *program generation* (or planning) in which the discovered programming plan knowledge is coupled with knowledge about the way in which target language programs achieve various programming goals within the context of the programming domain, and an

[3]For the purpose of this work we assume that the plan library is sufficient to *explain* any source that we consider.

equivalent program is produced in the target language. The remainder of this paper focuses on realizing the program understanding portion of the translation task.

A model library-based program understanding

The Quilici model (Quilici 1994) provides one specific expression of the process of program understanding. In this model, the discovery of programming plans is achieved through matching domain-specific and general programming-paradigm knowledge expressed as plans to the actual source, attempting to infer more general or encompassing plans[4] and specializing recognized plans[5] in order to narrow the scope of the source as much as possible.

In the interests of not bogging down this presentation with excess detail, an explanation of the Quilici representation of program understanding is provided in Appendix A.

The methodology outlined in Appendix A presents one hybridized approach to plan recognition within the structured domain of source code. Source code program-plan recognition has the advantage over traditional plan recognition domains that all of the *source to be understood* may be seen in advance of commencement of the recognition process. The Quilici algorithm merges a limited form of top-down, or context-directed mapping achieved through preferring to *specialize* matched high-level plans and to attempt to locate certain *indexed* plans as indicated by pre-specification of what plans are expected to share context with certain other plans with a standard bottom-up strategy which identifies individual "base" or source-level code parts and determines which higher level plans may contain or make use of these parts. Quilici relies heavily on indices and specializations that are explicitly defined[6], and consequently makes no provision is discussed for the [*inference* of possible explanations (upward) for sets of low-level plans other than described by specific indices.

The actual traversal of the Quilici search space has not been elaborated in any detail in (Quilici 1994), and the algorithm control strategies that would mandate such a search are not described. Consequently it is difficult to predict or examine the behaviour of such an algorithm to determine the efficiency of such an approach. What would be very useful is a formalization of this process so that an algorithm could actually be constructed and implemented, thus allowing for closer examination of the behaviour of any selected search strategy, or heuristic. This work is part of one step in encompassing all of the program understanding paradigm in this way. Quilici's approach constitutes a largely uncontrolled hybrid constraint propagation/search in the plan explanation space in which the potential explanations of each block-variable are limited with the continual addition of constraints as they are discovered through specializations, indexing, and instantiation to the plan library. Control issues such as which block-variable to focus on next, which explanation to adopt next when contextual clues are missing, and which templates are potentially "important" in

[4]Generalization is one form of bottom-up explanation.

[5]Specialization, is one form of top-down explanation.

[6]In fact, the algorithm assumes that any relevant connections will be specified, placing strong requirements on the quality of the plan library, and providing little flexibility for specification error or omission.

the plan library are ignored.

We have introduced program understanding as the first part of the translation task. First we have described the process of program understanding and examined one recent effort at modeling this understanding process which makes use of hierarchical knowledge about program plans in the construction of a model that explains program source code. This model reasons both from a top-down viewpoint guiding search for plans according to previous understood context, and bottom-up, constructing plans from essential plan components. In the next section of this paper we shall examine a formalization for plan understanding that is capable of providing for greater control of search by exploiting the existing knowledge structure in some limited ways.

```
int matchAll( char* type1, int locX1, int locY1,
                   char* type2, int locX2, int locY2,
                   int minRange, int maxRange)
{ if !( matchType(type1, type2) )
     { return -2; }
  else
  { if !( inRange(locX1, locY1, locX2, locY2) )
       { return -1; }
    else
   { return 1; } }}

int matchType(char* type1, char* type2)
{ if ( (substr(type1,0,2) == substr(type2,0,2)) ||
       (substr(type1,0,1) == ``*'') || (substr(type2,0,1) == ``*'') )
     { return 1; }
  else
  { return 0; } }

int inRange(int x1, int y1, int x2, int y2, int min, int max)
{ int d = sqrt( sqr(x1 - x2) + sqr(y1 - y2) );
  if ( (d > max) || (d < min) )
     { return 0; }
  else
  { return 1; } }

int matchOrient(int o1, int o2)
{ if ( (o1 > 8) || (o1 < 0) || (o2 > 8) || (o2 < 0) )
     { return -1; }
  else
  { if ( (abs (o1 - 02)) <= 1 )
       { return 1; }
    else
    { return 0; } }}
```

Figure 2: Geometric domain source code excerpt.

3 Re-formulating Program Understanding

Constraint Satisfaction Problems

A constraint satisfaction problem[7] (CSP) representation consists of three major components: A set of variables, a range of domain values for each variable, and a set of constraints amongst the variables restraining individual assignments of domain values to a particular variable depending on domain assignments to other variables. This representation corresponds to a graph in which nodes are variables, arcs represent constraints both reflexive (constraints on domain values or ranges), and inter-node (constraints *between* domain value assignments). A solution to a CSP is simply an assignment to each variable of a domain value from that variable's domain such that all constraints restricting assignments are satisfied. In some cases we may be interested in only a single satisfying assignment or solution, while in other problem instances we may wish to locate all possible solutions.

In the interests of not bogging down this presentation with excess detail, a high-level overview of CSP representation issues and solution strategies is provided in Appendix B.

Advantages of CSP representation

CSP representation and solution has provided a rich area of research over many years. A wide variety of problems have been successfully represented and solved in this paradigm. Languages providing for convenient problem representation as constraint problems have been constructed and are used in many domains. We have outlined briefly a small portion of the large body of recent work that has focused on constructing efficient search-based algorithms for solving CSPs. In earlier work (Woods 1993*a*) we describe a problem known as *spatial template recognition* and provide empirical evidence of the practical applicability of a variety of search and propagation algorithms in problems of real size. Elsewhere, (Prosser 1993) describes and evaluates in some situations the entire range of algorithms commonly applied to CSPs, and (van Run 1994) studies the ordering and re-ordering of variables in various CSP search strategies.

CSP solutions may be characterized by a variety of search algorithms, domain dependent heuristics that are used to control a particular search strategy, domain independent heuristics that essentially change the nature of one search strategy to another, possibly more CSP-specific strategy, and localized strategies that tend to focus on repairing a rough solution iteratively until it is "good-enough". We see the following issues of prime importance for any attempt to represent a problem as a CSP and subsequently to control search in a CSP based on the structure of domain knowledge:

- *Expressiveness/Representation* : can our problems be expressed concisely and "naturally" but also effectively in the language ? Does the problem expression provide sufficient important domain knowledge so as to suggest solution strategies ?

[7] See (Kumar 1992) for an accessible and detailed treatment of CSPs.

- *Search control* : can the strategy selected be modified dependent on the problem represented, or even possibly based on a given problem instance or dynamically based on problem characteristics? Specifically, what information can help guide this modification process ?

- *Consistency propagation* : as a precursor to search or in conjunction with search, it often appears to be sensible and beneficial to "propagate" constraints and limit one domain size based on a previous variable domain value selection. Typically, these propagations are done based on problem-specific characteristics (heuristics), or CSP-specific heuristics (such as domain size). Can these features significantly impact search in a given domain ?

- *Hybridization* : While both use and specification of search strategy and consistency propagation are important, many problems are best expressed through an interleaving or "hybrid" of the two methods. In fact, it has been claimed (Nadel 1989) that all search strategies can be expressed as an interleaving of some amount of in-search constraint propagation and standard backtracking (BT). From this viewpoint, Forward Checking (FC), LookAhead (LA), Partial LookAhead (PLA), and "intelligent" backtracking such as BackJumping (BJ) and BackMarking (BM) are all essentially primitive BT with a certain amount of in-search constraint propagation. Consequently, construction of interesting or novel search strategies depends upon being able to control (possibly dynamically) the amount of effort spent on consistency propagation at the expense of backtracking throughout the search space.

- *Domain independent/dependent heuristics* : Do there exist specifications about problem aspects that potentially make the problem "easier to solve" exploitable during search ?

Program Understanding as CSP

CSPs are formulated in terms of a set of variables, a set of domain values, and a set of constraints amongst variables dictating which values may be assigned to each variable. We construct our CSP model of program understanding with the following correspondences:

- *Variables*
  Program understanding as conceptualized earlier in this paper is concerned with explaining the purpose of a piece of source code by mapping the source code to existing knowledge about program plans. Source code consists of many parts, and an explanation of the entire code will be built of smaller, dependent explanations of these parts. The "part" we define as a functional block of code, easily determined as a function or procedure in the larger source. An explanation of this "part" in terms of our plan knowledge is essential to an understanding of the larger source. A variable in our

formulation corresponds to a functional block. The process of decomposing the source into functional blocks is referred to throughout the remainder of this paper as *blocking*.

- *Domain Values*
  Our variables corresponding to functional blocks have domains that range over all possible *explanations* of the design purpose or programming plan for that block. These programming plans are drawn from the programming plan library. With little specific structural or contextual knowledge, an arbitrary block could be explained by anyone of the many possible programming plans.

- *Constraints*
  Variables correspond to the functional blocks of our source code, each of which ranges in its possible assignments over all possible programming plan explanations in the plan library. Constraints embody all of our knowledge about the way in which programming plans may fit together. Explaining one functional block in a certain way is going to dictate how other functional blocks may be explained, according to our knowledge about how programs are constructed to meet program goals.

- *Solutions*
  A solution to this CSP is an assignment, from each variable to one of domain-values or plans such that no constraint imposed by the plan structure is violated. It is important to note that there possibly exist "solutions" of varying degrees of specificity. Since our domain-values are in fact plans which are potentially "abstract" entries in the hierarchical plan knowledge structure, an abstract assignment corresponds actually to a "possible" specialization of that plan. In fact then, an abstract instantiation of a block-variable corresponds only to a narrowing of the range of the variable with respect to the lowest-level possible plan. Different solution strategies suggest themselves in this type of hierarchical domain-value environment. One such approach involves solving a problem at an abstract level across the set of block-variables as suggested in the hierarchical planning work described by (Yang, Tenenberg & Woods 1992, Yang, Tenenberg & Woods 1994). Another approach would be to allow for solutions at varying degrees of specificity across the block-variables set during search. This type of approach requires the re-formulation of search to reflect that an instantiation is only a domain reduction of a particular variable, and that another choice point in search is the specialization of a particular abstractly instantiated variable.

  In (Mackworth, Mulder & Havens 1985), the construction of consistency propagation algorithms for hierarchical structured domains is discussed. Depending on the nature of the hierarchical domain decomposition, the Hierarchical Arc Consistency (HAC) algorithm has been shown to have a worst-case performance that can be between twice as bad as standard AC-3 and exponentially better[8]. The key to acquiring this

[8]Specifically, the worst (time) case of AC-3 is known to be $O(2a^3e)$ where a is the initial domain size, and e is the number of constraints in the problem, and the worst case of HAC is $O(4a^3e)$. However, when

performance increase is that the domain value hierarchy (plan library) specify groupings of "natural kinds", that is, a strict downward reduction from general to specific.

Given the structuring of recognition as a CSP reduction and search problem, it is possible to utilize any known constraint satisfaction search algorithms or constraint propagation algorithms, including potential incorporation of HAC as a strict downward refinement tool during reduction or constraint propagation. We next detail the incorporation of constraint and library structures into the CSP formulation in somewhat more detail.

Encompassing source structural relationships

We are interested in structuring understanding as CSP so as to profitably exploit specific domain knowledge in a reasonably well understood framework. Some of this domain knowledge is embedded in our programming plan library formulation, but additional structural information in the source itself can be utilized in the mapping to the plan library.

In addition to a parsed interpretation of source as an abstract syntax tree, our construction of the source as "functional blocks" provides for another level of structure. We make the assumption based on common programming practice that a functional block in the source code will correspond to some set or individual programming plans. In order to further constrain the nature of these blocks, we may identify block characteristics, such as the type of input parameters, and the type of return values. While such functional blocks may have side-effects outside of a well-structured input and output paradigm, we also assume that the *primary plan* of the block is reflected explicitly with respect to return value(s) or output parameters. In addition to "entry" and "exit" structure, functional blocks have relationships with other blocks. Each block has spatial (nearness) or functional (call/called) relationships with other blocks that will be used to direct search for a solution. These spatial relationships may be viewed as "soft" or preferential constraints on potential block-variable explanations, while "functional" relationships are harder constraints. Intuitively, the notion that programmers tend to group functionally related blocks close together is captured in these "nearness" constraints. In contrast, the call/called relations indicate definite functional relationships that need to correspond to program plan design knowledge. Figure 3 provides a depiction of the functional blocks with relations annotated as derived from the code shown earlier in Figure 2 on page 10.

Interpreting Figure 3 as a first-cut at a CSP representation, each block is a variable in need of explanation, with the constraints specified indicating that (1) an explanation of any block (say block 1) "physically close" to another block (say block 2) is likely to satisfy the property that the block 1 explanation and block 2 explanation are conceptually closely

the domains are "appropriately structured", that is when additional evidence about related variables results in a necessary reduction in the size of a variable's domain set we can say that the specialization hierarchy is used as a discrimination tree, and in this case the worst case of HAC is only $O(loga \cdot (e + 3/2n))$, for n variables.

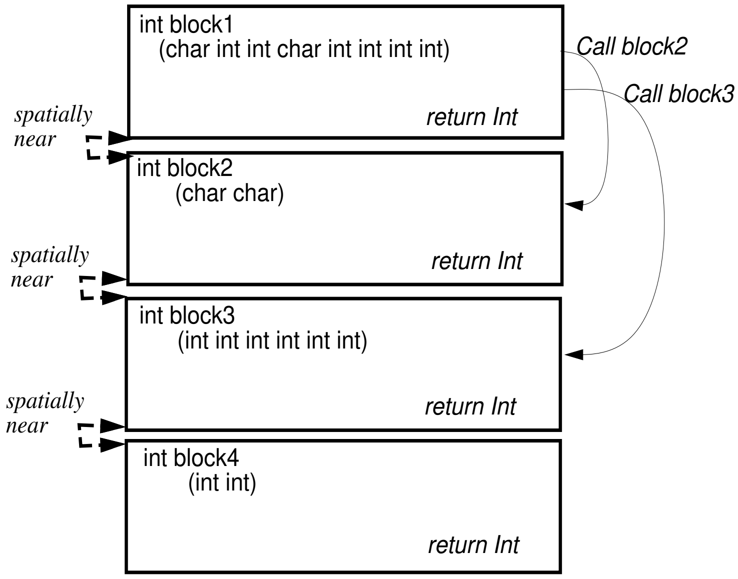

Figure 3: Source code blocks with inter-block constraints shown.

related as parts of some larger, encompassing program plan, and (2) an explanation of a block (say block 1) that calls another block (say block 2) must account for the calling relationship in terms of program plan knowledge, and also account for the type information of the call. Additionally, the extracted "boundary" information of call and return types can serve as "node constraints" which restrict the range of possible plan explanations applicable to that variable-block.

Interpreting plan library knowledge

The plan library hierarchy example of Figure 1 forms a set of implict inter-variable constraints on the allowable variable/plan explanation assignments as embedded in Figure 4. The plan library can be seen as constraining all pairs of variables according to the structure of the library. For example, selecting a plan assignment of some plan P_1 for block 1 and some plan P_2 for block 2 would require a check of the constraint "block 1 calls block 2". This check is accomplished by searching for this call/called relationship in the plan library entries for P_1 and P_2. The absence of such a relationship results in a failed constraint check.

There are two types of primary plan relationships detailed in Figure 1, specialization and composition. The specialization relationship is basically an "IS-A", where some abstract programming plan has one more realizations. For instance, an **Attribute-Match** can be either of a **Type-Comparison**, a **Orientation-Match**, or a **Location-Match**. Specific methods for testing the specialization from abstract to specific may be represented in the "IS-A" box of the figure. These methods describe tests for discrimination among possible specializations. A composition relationship specifies that an abstract programming plan is constructed of several major components in a certain way. For instance, A **Type-Comparison** is constructed of a **Global-Star Match** and a **Type-Key Equality check**, and the "consists-of" or composition relation is potentially specified in some detail, for instance describing that the larger block of code embodying the **Type-Cmparison** plan in fact *calls* both of the subordinate program plans. Additional information about the relations between a "parent" and its composite children may be also represented in this fashion. The spatial relationship, call/called, and block boundary constraints are detailed in Figure 4 along with the implicit constraints derived from the plan library.

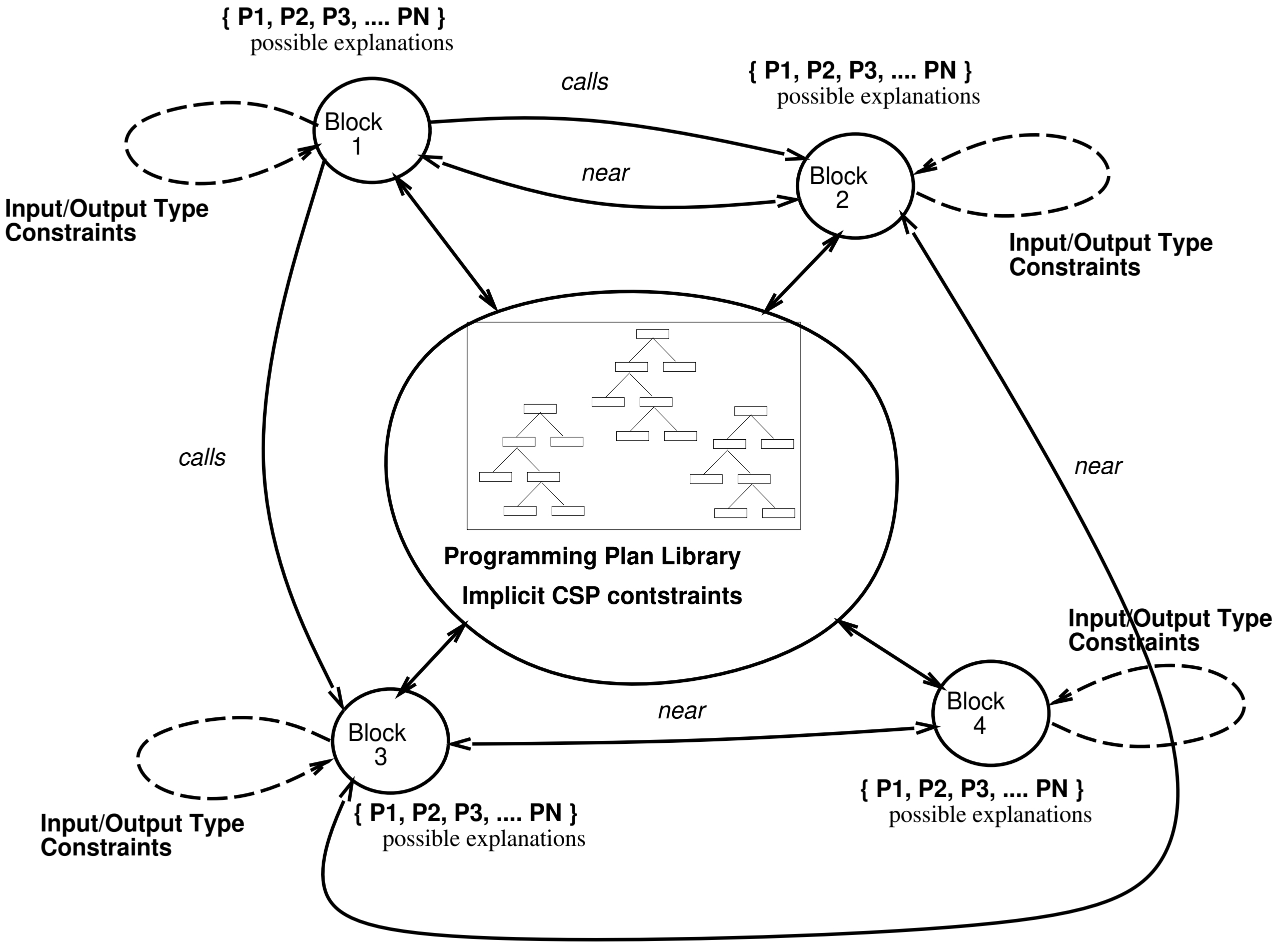

Figure 4: Program Understanding CSP : plan knowledge as constraints.

In addition to forming a basis for constraint application, the plan library is also the source for the block-variable explanation ranges. We require an initial assignment of all possibly-explaining plans to each block-variable. The library consists of many plan *template* occurrences arranged according to known domain plan structures. For instance, a programming plan template that describes a general function such as *Iterate-Over-List* will potentially appear as a component plan of many other complex, abstract plans. In Figure 5 we see how a particular template might have many occurrences in the hierarchy. Each template that *possibly* maps to a block-variable signals a candidate *set* of explanation plans for that block-variable, with each occurrence of that template in the plan library being selected as a candidate. The initial identification of block-variable range is based upon an interpretation of the type-boundary of the functional blocks, allowing for at least some initial reduction of the size of the problem space.

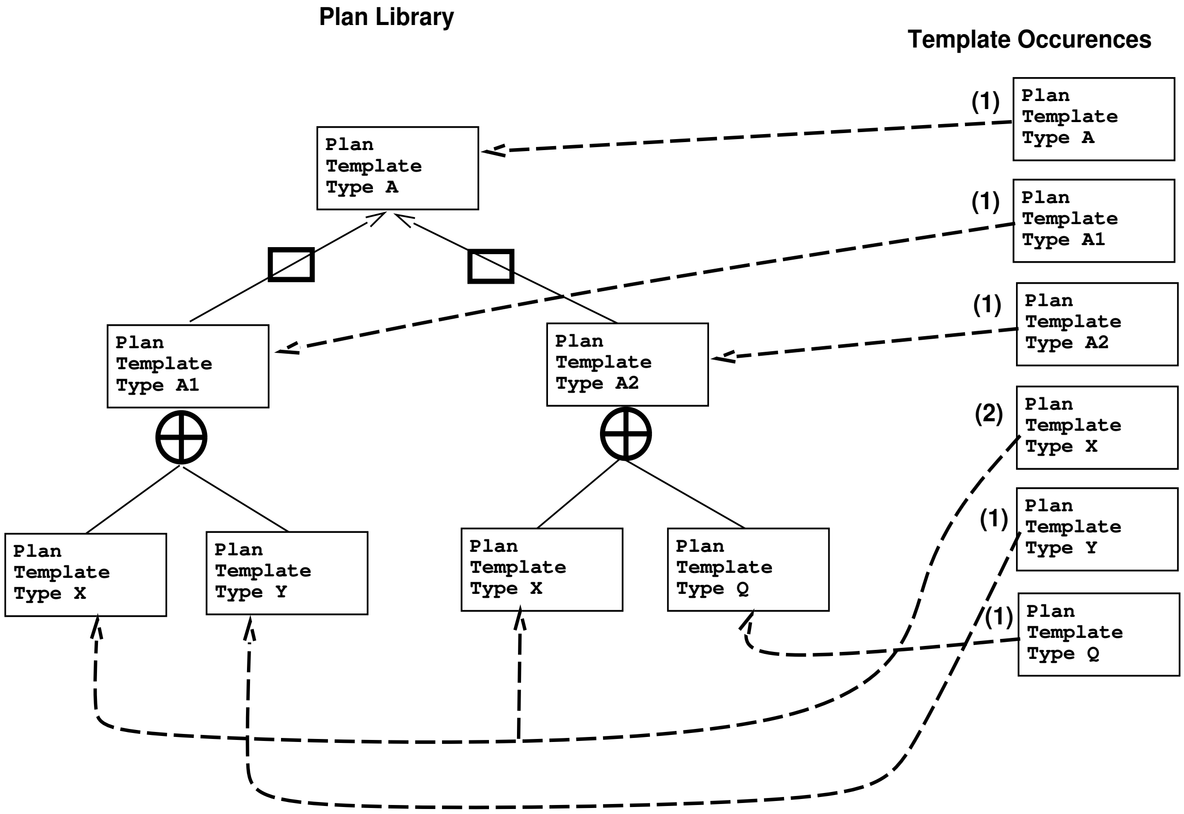

Figure 5: Plan library template occurrence index.

Given a situation with variables each ranging over a certain range combined with a set of inter-variable constraints dictating allowable variable instantiations, we now have a problem that is accessible to the range of constraint tools described earlier. In the following subsections we describe several ways in which the knowledge structure of our plan library and the existing source structure can be adapted to serve as heuristic controls on problem solution. Our future work will be aimed at determining experimentally the circumstances and problem types under which certain combinations of these heuristic approaches will be of most benefit in typical program understanding tasks.

Task planning as variable-block ordering or selection

We are interested in finding explanations of our set of variable-blocks. This combined explanation is a potential explanation of our program source. We would expect to find only one such solution. Consequently, we are interested in traversing the search space with the goal of identifying one solution. Viewing the problem as one of search, several issues are critical for efficient performance. We wish to guide the search in such a way that the already explained blocks result in a narrowing of the possible number of explanations of the remaining blocks. Essentially we wish to *focus* on the blocks that are most restricting earlier in search. Essentially, we are going to plan which aspects of our multi-part task to work on first, and which to postpone. If we view this in CSP terms, we are going to spend some effort at determining which order in which to attempt to instantiate variable-blocks.

Good variable selection has typically been attempted so as to restrict the branching factor of search, that is, to select a variable that is well-constrained in terms of having few potential domain-values (explanations), and which may also have a large constraining effect on other variables (blocks). Consequently we can see that if a block corresponds to *few* potential templates, and then, to *few* potential template occurrences, and also if these occurrences themselves are related to or are a part of structure in the plan library that are *unique* or known to occur relatively infrequently, then this block is a good candidate for selection or focus of effort.

We have then several obvious measures that can be made from the plan library structure and exploited in subsequent CSP solution. A domain value at an arbitrary level of abstraction is actually a template with several instances in the library. Further, if this template (at each instance point) is abstract, it may have several potential refinements to other template types. Some examples of these types of plan library measures include:

- *Block width* - a particular variable-block may correspond to several templates. A variable that has fewer possible template mappings is less ambiguous and thus carries more restriction to search if selected.

- *Block overall width* - a summation of the number of possible instances of all possible template matches to the block. A block with fewer possible explanation instances should be preferred since at this point a lower branching factor would be introduced with its selection.

These types of measures about template instance frequency are another source of knowledge that can be viewed as a "soft" or preferential constraint extractable in advance from the knowledge library.

Task planning as domain-value ordering or selection

The measures we introduced for block-variable selection or task planning have actually a more specific application as well. Once a block-variable is identified as a current focus, the selection of an actual plan explanation must be made next. A determination of which *template type* of several possibilities can be made on the basis of correspondence to existing source context (such as suggested by preferential constraints extracted from source such as spatial relationship or call/calling context), or from existing "understood" context (such as likely gaps in our mapping to the plan library that should be filled in based on added common ancestor constraints). This decision constitutes a narrowing of the domain to a particular subset of all possible base-level or leaf plans. This decision may be made on the basis of overall width as described earlier (fewer explanations one level down is better), or possibly absolute width (where fewer base-level or leaf explanations is preferred). In either case we exploit our knowledge about how a particular selection may be later refined or contextualized.

Upward explanation as greedy-contextualization

Given two variable-blocks instantiated with plans at some degree of specificity, and given that these blocks are somehow 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. Thus, identification of this Occam parent template and then selection of a variable-block that is potentially instantiable to this template type is a reasonable task plan. In essence, an inference is made about the context the two plans share and an attempt is made to control search *top-down* based on this existing context. The determination of such a relationship is realized through a *dynamic constraint addition* which constrains the two siblings to be related as descendants of their common ancestor. In order for these two siblings to take on another relationship, this constraint would have to be removed via backtracking. A subsequent decision based on this identified common parentage would be to suggest other possible plans that belong in this context (such as a third low-level sibling to the two instantiated plans).

Template matching as greedy contextualization

In contrast to identifying common parentage and thus controlling search top-down via task planning, it is possible to analyze the plan library further in advance of search, determine *unique* and subsequently important plans whose existence in the source would dramatically narrow the possible context of the entire source. For instance, in a particular domain if some template occurs very infrequently in the plan library, however is known to occur frequently in practice in our particular domain, then a good strategy is to determine initially whether or not such a template is present in our source. If this plan **is** present, then the instantiation of a particular block-variable accordingly is a very constraining decision. If this plan *is not* present, then potentially other *high-payoff* scans can be made prior to additional search. The analogy to expert behaviour this approach displays stems from other work which has shown that readers scan ahead for familiar features and words as "clues" when reading pages of text. For program understanding, experts quickly scan source to determine if it is something *like* something they are familiar with and have had experience with. If it is, they attempt to contextualize the entire source within this framework through instantiation of a particular block-variable and expanding this newly discovered context to block-variables "related" or "preferred" by the matched block-variable. This scan for uniqueness is represented as an attempt to match several *well known* templates as determined from their occurrence in the plan hierarchy, and if possible, information about their rate of incidence in past program understanding tasks. Few template occurrences in the library combined with a high incidence in understanding suggests an important "identifier" plan.

As an example of a progam plan template formulated as a template recognition problem in the spirit of the spatial templates described and recognized as described in (Woods 1993*b*), consider the program fragment shown in Figure 6 which is represented abstractly as a template in Figure 7. It is interesting to note that in representing the code as a template it is necessary to make explicit much of the implicit representational structure of the code excerpt. Additionally, this template represents an abstraction in which an array is accessed by a function one element at a time sequentially, but starting from either the head of the array or the tail. The task of formulating this explicit representation is an interesting topic for future work, but is assumed for our purposes to be expert-constructed.

```
for( int cnt = 0; cnt <= size(A) ; cnt++ )
{
 /* other code may exist here */

 SomeFunction( A[cnt] );

 /* other code may exist here */
}
```

Figure 6: Iterate-Apply-Over-Set program plan template instance as *C* code.

In (Woods 1993*a*), experimental results are cited in which spatial templates of this complexity are identified in spatial situations in which *noise* in the form of false-possible matches to each feature-variable is added. These results show that despite the addition of noise that overwhelms a traditional expert visually, a CSP formulation solved with simple backtracking algorithms such as Forward Checking, or backtracking combined with preprocessing of the search space for arc consistency with the AC-3 algorithm was able to extract *all possible* satisfying mappings from template to situation elements in a short amount of time bounded by several minutes.

In Section 4 we outline some similar experimental results for identifying source code segments that embody specific programming plans.

Summary

Selection of block-variable or variable-ordering in search can have dramatic effects on search performance (Nadel 1989). Constraint propagation algorithms that reduce the number of potential domain-values or explanations for a particular variable or set of variables have been shown in other domains to reduce overall search effort. This selection is a top-down control strategy insofar as the next focus is the issue of concern. Bottom-up controls are also important. For instance, evaluation of the plan library to extract the "key" templates which are most informative for matching can assist in context-greedy approaches. Essentially, if a unique template is located, the remainder of search can be highly constrained, at least in the space local to this variable-block explanation.

Given the existence of reasonably efficient methods for recognizing specific instances of code templates in large bodies of code, it is easy to envisage adapting known spatial template recognition approaches to function as first-pass context-greedy search-limiting methods. The fusion of such recognition as part of a larger program understanding methodology is the basis of our research of which this work is only part.

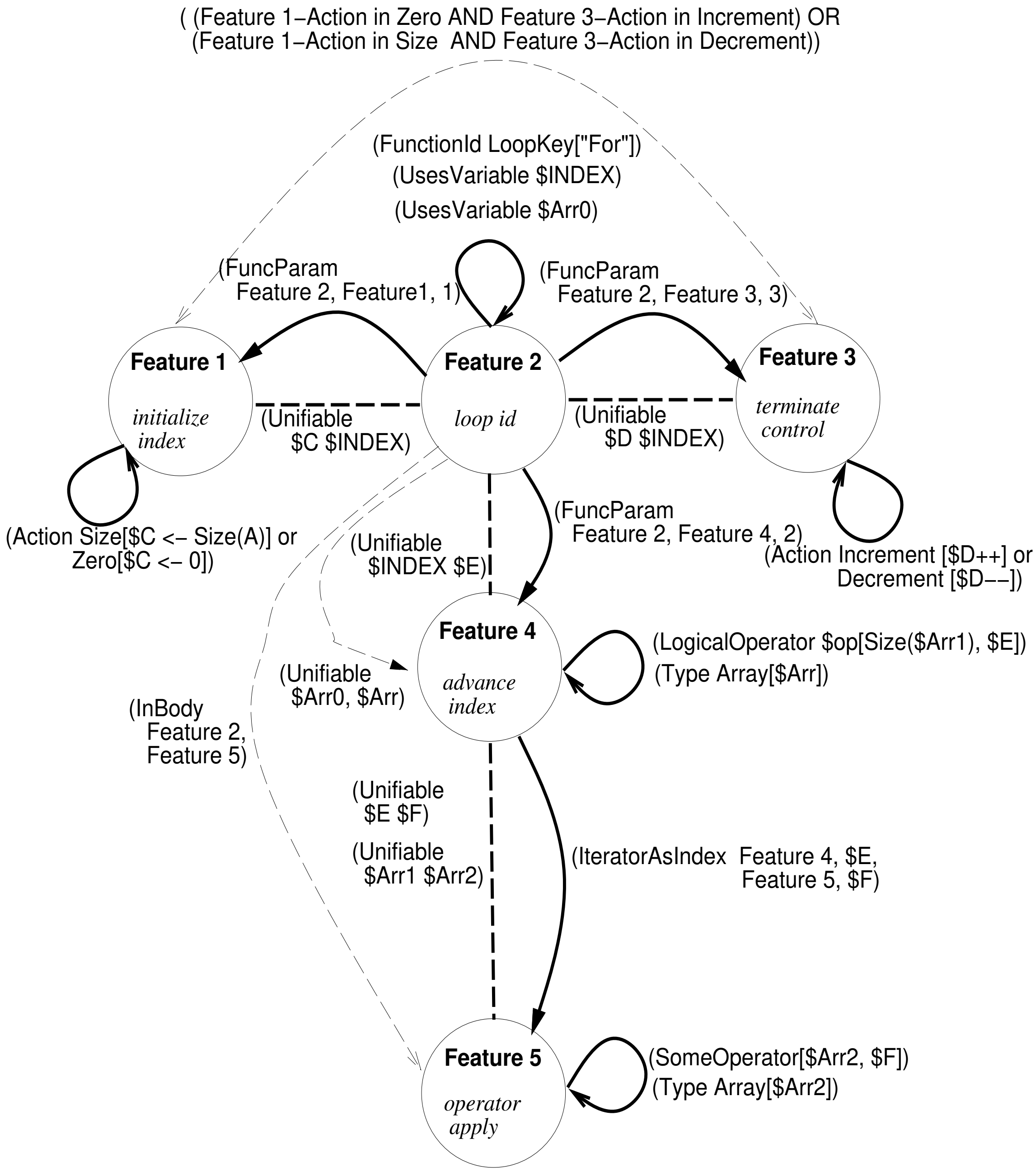

Figure 7: Program plan template with 5 feature variables, 11 arc constraints.

4 Experimental results

Introduction

In this section we describe some early experimental results that demonstrate through several simple examples the feasibility of recognizing sets of source code that we wish to encapsulate within the framework of an existing abstract datatype. This abstract data type (or object) embodies data and function, and we wish to identify (and replace) code in our source which should make use of this data type, but which does not. We demonstrate empirically how this local plan identification (in this case, abstract data type identification) scales up over larger source cases.

Overview

In the previous sections we have described program understanding as one part of the task of translation between source languages. This task was formulated as a constraint satisfaction problem with the purpose of integrating known search and constraint propagation methods for CSPs with novel approaches at exploiting knowledge structured as a programming plan hierarchy in reducing search. As an effort at constraining the size of the range of explanation of source code block-variables we described the use of model template recognition techniques as part of an attempt to form some initial contextual clues into which the remainder of the program source may be understood. We gave an example of a code template constructed specifically to be of the same relative complexity as previously examined spatial templates.

Our previous outline of program understanding as CSP envisaged the program plan hierarchy as represeing actual source code on the leaves of the "tree" of plans. We encapsulated the intent of source modules within the interior of the library and then proceed to search for possible plan matches. In some knowledge representations we wish to represent both function and data within objects. These objects essentially meet program design goals and consequently fit as source on the leaves of the tree as instances of higher level goals. We will wish then to be able to map or match "abstract" versions of these type of "object" or abstract data types within the context of source code written without the benefit of these types. This abstract data type embodies both *data* and *function* aspects.

We consider for the purposes of our experiments a source program which has a primary function of generateing several different types of geometrical objects on a display device. These geometrical objects are constructed through a series of "printObject" function calls. We are given a program plan library including these abstract data types and wish to identify several specific ones as indicated by the larger program understanding task, which we assume.

In Figure 7 we defined a source code template in terms of the composition of its primary code components. In (Woods 1993*b*, Woods 1993*a*) templates are defined geometrically based on the relative spatial relationships of their primary components. In the domain of geometric abstract data type recognition, it is possible to define a template either based on some abstraction of its code composition and function, or as a geometric configuration with direct mapping to the source which created the configuration. The later is a case of making use of *program output* in the understanding task. Earlier we identified the information source as consisting of the source code partitioned into blocks, with initial block relationships such as spatial distance and functional interactions forming a primary part of the CSP constraint set. In this extended example, the program output generated by individual source code lines is also used[9].

With a spatial template representing the geometric output of an ADT, the ADT could be matched against the source program output such that a match would indicate which sets of source-code-produced geometric objects represent parts of the ADT in question. Determination of the source statements or constructors of the displayed objects would complete the mapping, and these source constructors would be identified for replacement with call or use of the ADT functionality. Initially we consider that this second stage would be performed by an expert, and the purpose of the initial stage is the identification of the ADT instance.

With templates representing ADT code, a slightly different situation exists than with geometric templates. In this case, rather than explicit realization of ADT parts, we have an "input" format of (say) "printObject" commands. These commands realize on an output medium the objects that we referred to in the geometric realization approach. Rather than having explicit spatial relations that can be "measured" geometrically the input format has positioning variable values such as indicated by "printObject typeX at (x,y)". The computations, however remain the same. A "print" statement merely reflects the object it prints in a different format than a geometric realization. For the purposes of this experimental section, and in order to minimize the amount of new code produced for these experiments, we present our results obtained from a geometric interpretation of an ADT matching problem. In future extension of this work, an entire domain will be constructed for the program recognition problem.

The Abstract Data Type (ADT) domain

Figure 8 details an abstract data type for a particular, arbitrary[10], geometric object configuration labelled here **WarpCross**. This data type has five attribute slots, slot 1 of type square, slot 2 of type circle, slot 3 of type square, slot 4 of type triangle, and slot 5 of type trapezoid. We wish to locate instances in source code where this configuration of base objects is created, and identify these instances together as an instance in which the **WarpCross**

[9]In addition, a further use could be to examine the *temporal* relationship of functional blocks to determine not only the calling/called structure, but also on a larger scale, the frequency and repetitive cycles of a given set of program runs. This is one aspect of our future research.

[10]In fact, we intended to create an ADT that is complex enough to provide a range of constraints to check, and constrained enough so that there were not thousands of "phantom" solutions. Also, we wanted to create a data type that had strong analogy to the programming plan template we introduced earlier, so the ADT template has an identical constraint structure.

ADT functions can replace the independent constructions. The task is to recognize a *set* of source functions as related within the boundaries defined by the WarpCross ADT definition which defines the applicability of the WarpCross functionality in this domain.

Figure 9 depicts in the form of a constraint graph the WarpCross ADT. In the graph, the ADT constraints are represented as spatial constraints between objects for separation and alignment.

```
Define AbstractDataType WarpCross
WITH
 /* slot and slot attribute list */
 slot1: square of    size medium, orientation=\$o1, color shaded, AT x1,y1;
 slot2: circle of    size medium, orientation=\$o2, color shaded, AT x2,y2;
 slot3: square of    size medium, orientation=\$o3, color shaded, AT x3,y3;
 slot4: triangle of  size small,  orientation=\$o4, color $c4,    AT x4,y4;
 slot5: trapezoid of size large,  orientation=\$o5, color grid,   AT x5,y5;
SUCH THAT
 /* membership constraints */
 LeftOf(slot1,slot2), Separation(slot1,slot2,min=5,max=7),
 RightOf(slot3,slot2), Separation(slot3,slot2,min=5,max=7),
 \$o1=\$o3,
 Behind(slot4,slot2), Separation(slot4,slot2,min=2,max=4),
 InLine(slot4,slot2,maxVariation=2),
 Behind(slot5,slot4), Separation(slot5,slot4,min=4,max=6),
 InLine(slot5,slot2,maxVariation=2)
ACTIONS
 /* create an arbitrary instance of this type */
 WarpCross CreateWarpCross(int centreX, int centreY, ... ) { ... }
End ADT;
```

Figure 8: Code representation of a geometric abstract data type.

The source code input which generates geometric configurations (some of which may in fact be instances of our example ADT) is characterized by the source of Figure 10. Figure 11 is a geometric interpretation of one ADT instance as potentially generated by the code fragment. In this example the source is shown as creating the objects in slot order, however this is only demonstrated this way for illustrative purposes, and it is assumed that they appear in no particular order, and that the noise separating them is arbitrary. Ideally, we would like to recognize and replace the *sets* of **print** statements that generate an object configuration that is generatable through use of the abstract data type in question.

Test Examples

We wish to demonstrate the effectiveness of identifying instances of this ADT in source situations that are very complex. For our experiments, the source code is generated by first creating one set of geometric objects that satisfy the constraints of the ADT instance. Next, additional objects are created, more objects signifying more "noise" and consequently more source code. A sample of the geometric ADT instance of Figure 11, positioned on a 100x100 display grid is shown in Figure 12. In this representation[11], the **square** a objects are represented by the "plus" symbol, the **circle** objects by a "sideways box with dot" symbol, the **triangle** by a cross, and the **trapezoid** by a "box with a dot". In addition to a complete ADT (in the centre) there have been placed two "near fits" to the ADT, one with the two squares, and one with all five elements but not satisfying the ADT constraints. Initially with zero noise, the problem is relatively simple, since the number of objects that may fit each "slot" are few, and consequently the number of potential ADT instances is small. As noise is added, however, the ADT construction progressively "fades" into the background making it harder to distinguish from the other objects. To see this fading, consider Figure 13 which is the same situation as Figure 12, with the addition of 300 geometric elements as noise. The mapping from this to a source code example would result in source code in which 300 object creation statements are added, separating the five relevant creation statements arbitrarily. We have created as noise, statements which create objects of random attribution, thus resulting in many *possible* correspondences during search that must be removed from consideration through constraint propagation.

All experiments[12] have been repeated with the same set of 10 random distributions of attributes and locations. The results charted are the *worst case*, *best case*, and *median* (where median is the average of the 5th and 6th values when sorted) discovered in each of the 10 cases. In all graphs cases that exceeded 600 CPU seconds are charted for that result, or constraints evaluated at that point.

Search without advance *arc* constraint propagation

Figures 14, and 16 show the results in total constraint checks (work expended) of identifying all (there is only 1!) data type instance in varying amounts of source noise: from 100 to 1000 additional program statements arranged randomly in the source input. Search strategies represented are Backtracking, and Backtracking with Forward Checking and Dynamic Rearrangement[13]. Figures 15 and 17 detail the same problems, but refer to the CPU seconds

[11]Created via Gnuplot to avoid creating an entire system to create graphical output!

[12]These experiments were carried out on a SUN SPARCserver 470 workstation. Subsequent re-tests of selected data suggest an improvement time factor of between 3-5 on a SPARCserver 1000.

[13]Note that we also conducted experiments using BackJumping and BackMarking, however these results lie between the BT and AC-3 results and differ primarily in that they demonstrate less variation on average when compared to BackTracking.

required in place of constraint checks[14]. Node consistency (NC) checking is done in advance in all test cases, and time charted is total CPU time including node consistency propagation time. Where constraints are charted, this includes all constraints evaluated *after* node constraint evaluation. In the case of these search only strategies, the constraints reflect the amount of constraint checking during search.

Search with advance *arc* constraint propagation

As in the previous examples, Figures in this section show the results of identifying the ADT instance as placed in varying amounts of source code noise. In these examples, Prior to commencement of search, the CSP was simplified using the AC-3 arc-consistency algorithm. The constraint checks measured reflect the constraint checking done both during search and during execution of the propagation (AC3) algorithm.

We note that the result of applying NC and then AC-3 to these problems resulted in the location of all solutions via later search trivially. The propagation essentially located the solutions independent of search, and for this reason, Figures 18 and 19 summarize the amount of AC work done combined with only a minimal amount of search. The search work is essentially nil regardless of strategy in this case. It is conceivable that the optimal amount of propagation is actually less than that performed by AC-3, but we leave that question to future experiments.

Experimental Observations

From our experiments we were able to ascertain the feasiblity of this template recognition, even in larger sources[15]. Translating these results to the more powerful SPARCserver 1000 implementation we have a result of approximately 20 seconds using the AC-3 algorithm to locate *all* object configurations that meet the set of ADT constraints. A more localized search amongst just 200 "noise" objects results in an average CPU expenditure of around 1 second. Clearly this process does not need to be relegated to long-term batch processing when utilized as part of the program understanding task.

An interesting result was that the use of AC-3 in place of even intelligent search strategies showed a marked lack of *variation range* over the sets of 10 tests at each noise level. The use of other hybrid algorithms could potentially reduce the variation even further. This is an excellent result for the practical reasons that predictable performance in interactive systems is an extremely desirable feature!

[14]We present both in order to avoid criticisms of bias as a result of effort spent avoiding constraint checks during search or propagation

[15]It should be noted that no spatial indexing for locality was utilized during these template recognition tests, although such use could dramatically reduce the amount of work being performed by quickly eliminating potential domain values during search through a pre-compiled filtration technique. This reduction is also relegated to work-in-progress.

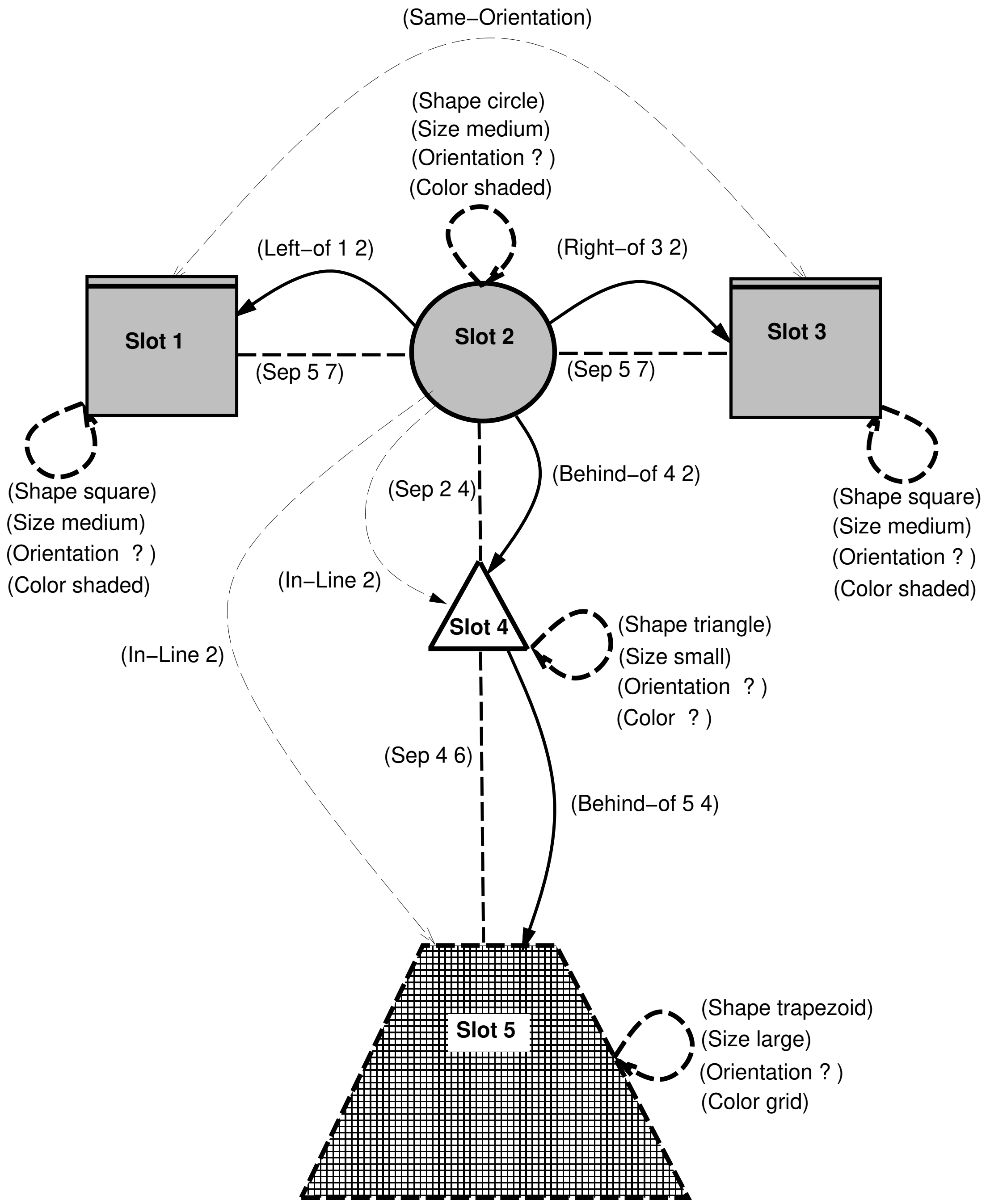

Figure 9: CSP representation of a geometric abstract data type.

```
printObject( ... );  /* noise */
printObject( Square, medium, 45left, shaded, x1, y1 );
printObject( ... );  /* noise */
printObject( Circle, medium, up, shaded, x1+5, y1+1 );
printObject( ... );  /* noise */
printObject( Square, medium, 45left, shaded, x1+10, y1 );
printObject( ... );  /* noise */
printObject( Triangle,  medium, up,      clear, x1+6,  y1-1 );
printObject( ... );  /* noise */
printObject( Trapezoid, medium, 90right, grid,  x1+1,  y1-7 );
printObject( ... );  /* noise */
```

Figure 10: Geometric generative code, including ADT instance.

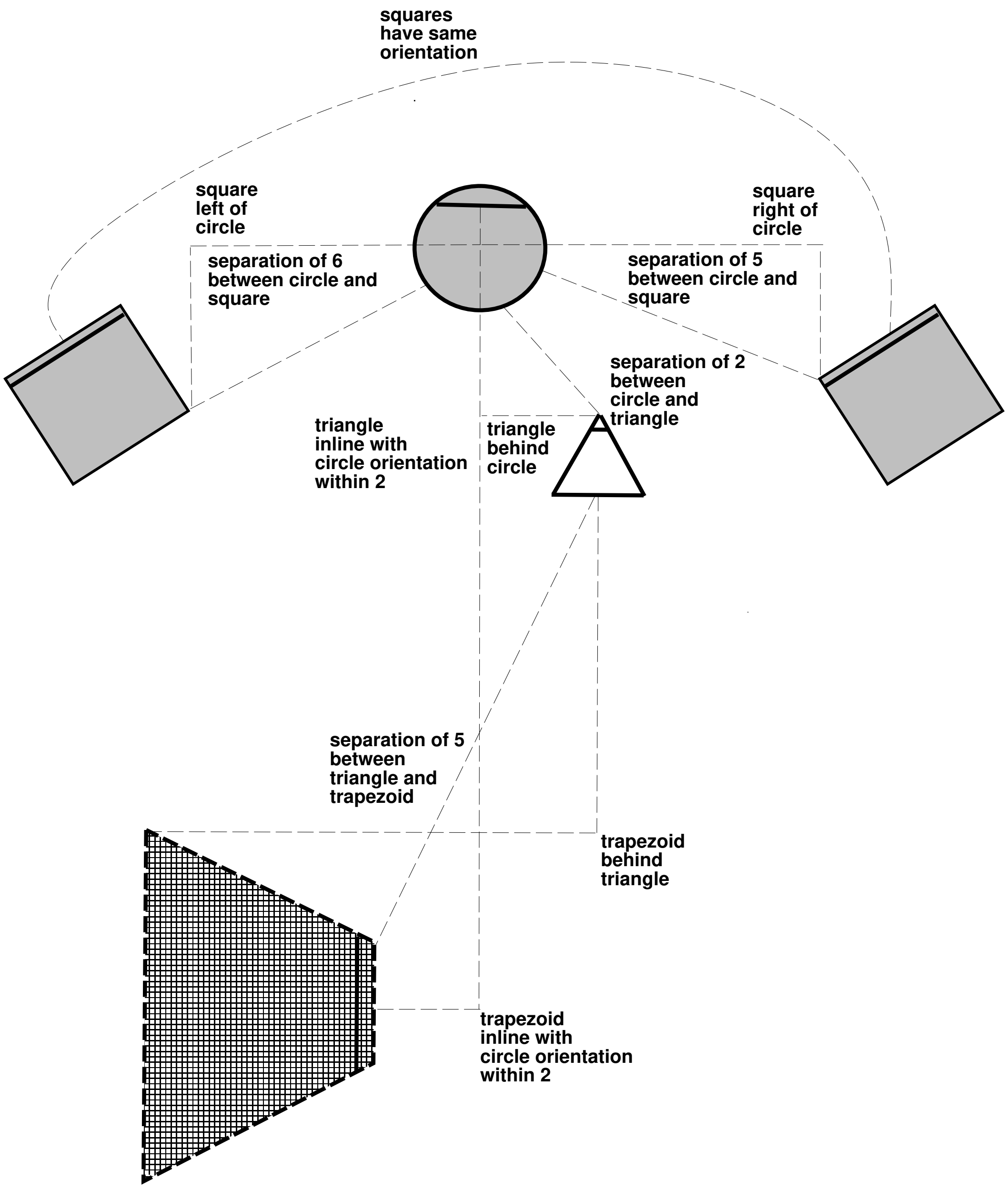

Figure 11: Graphical representation of satisfying ADT instance.

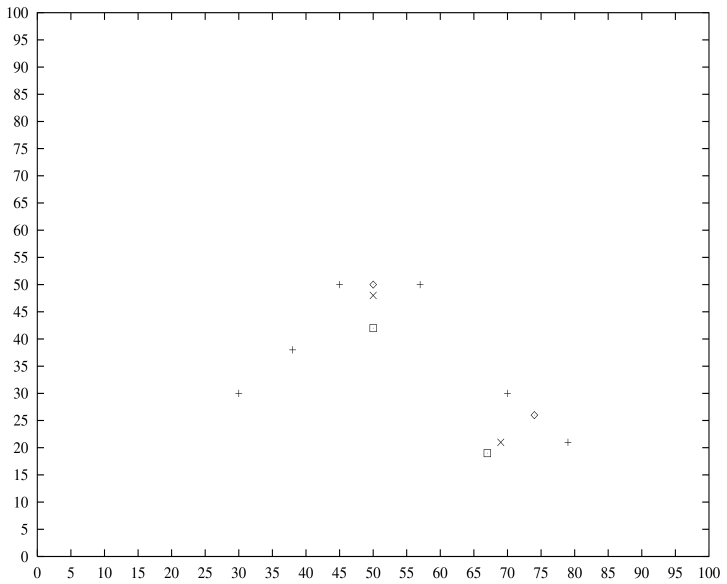

Figure 12: 1 complete and 2 partial WarpCross geometric ADT output structures.

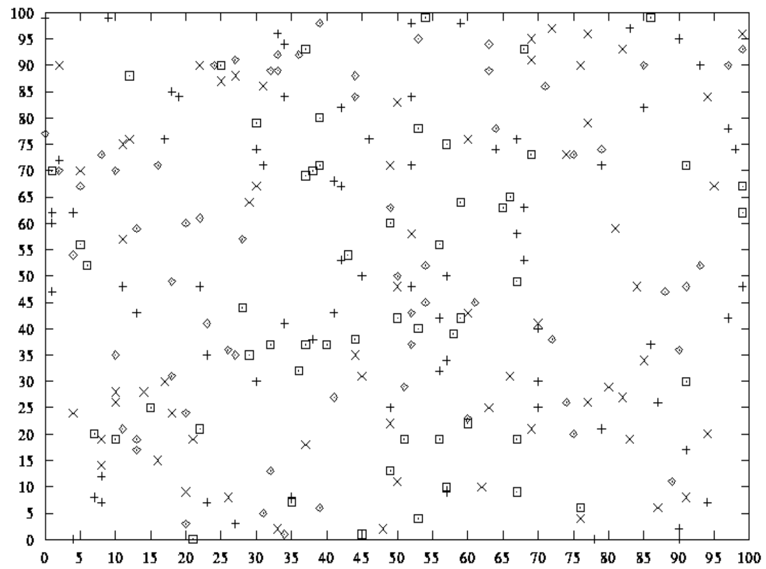

Figure 13: An output situation including 1 ADT and 300 geometric units as noise.

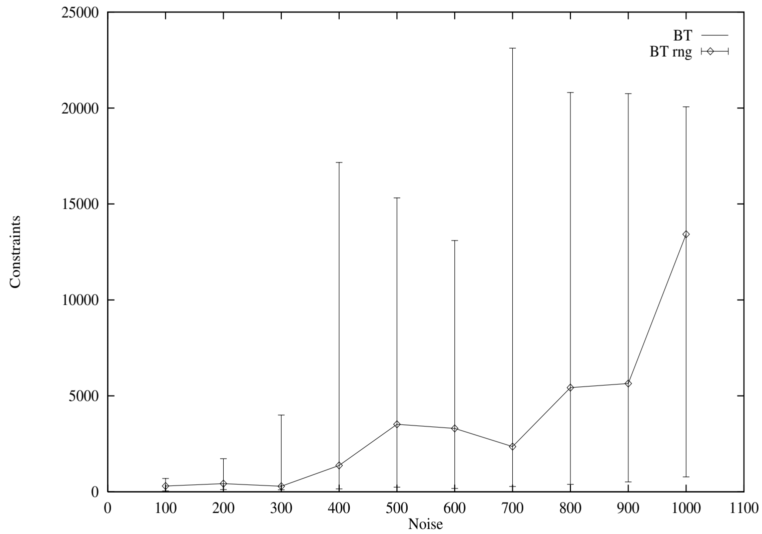

Figure 14: BackTrack constraint checks.

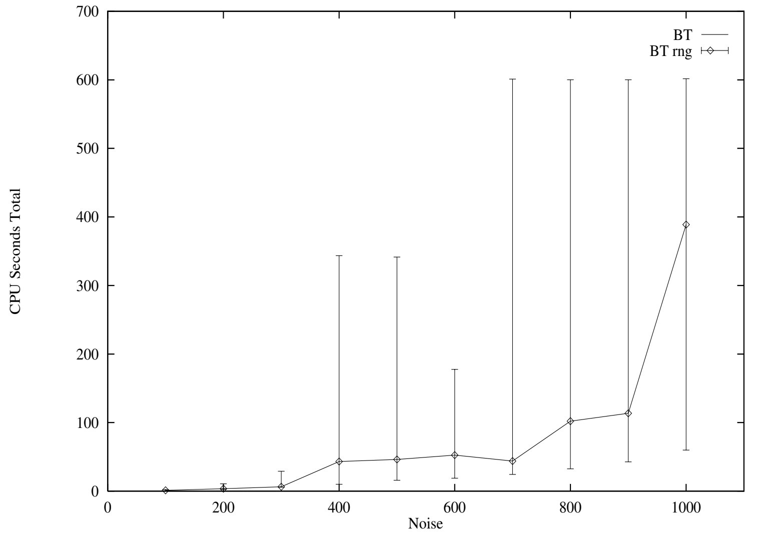

Figure 15: BackTrack CPU seconds (total).

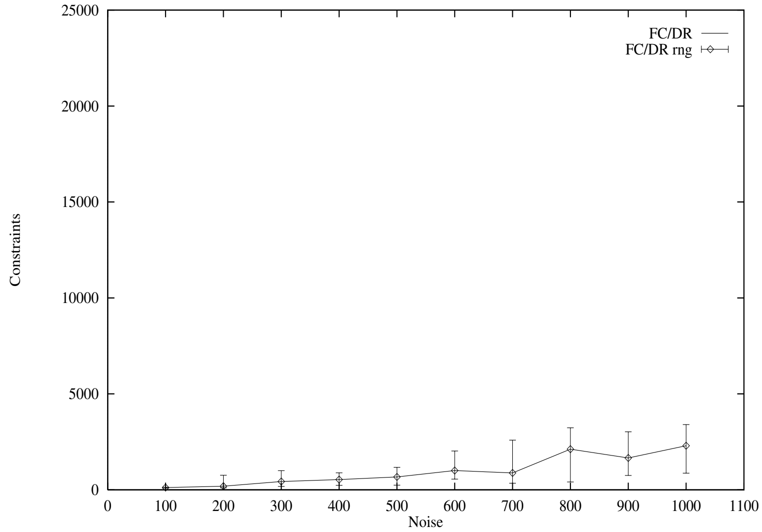

Figure 16: FC/DR constraint checks.

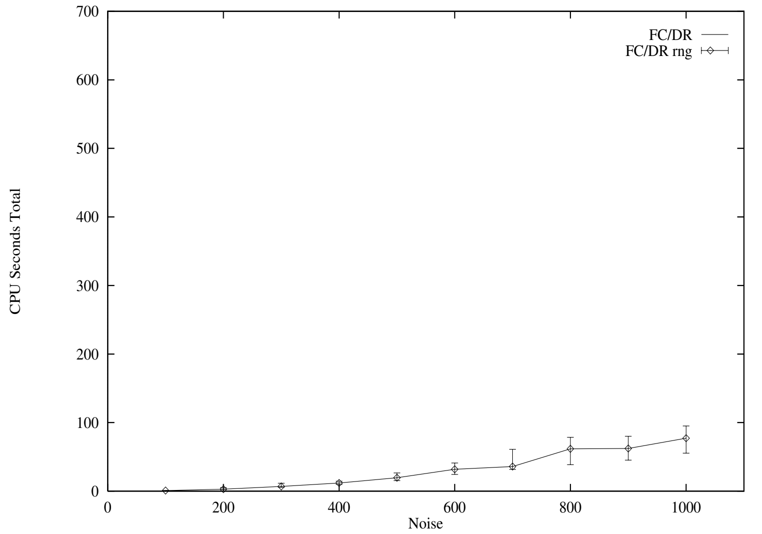

Figure 17: FC/DR CPU seconds (total).

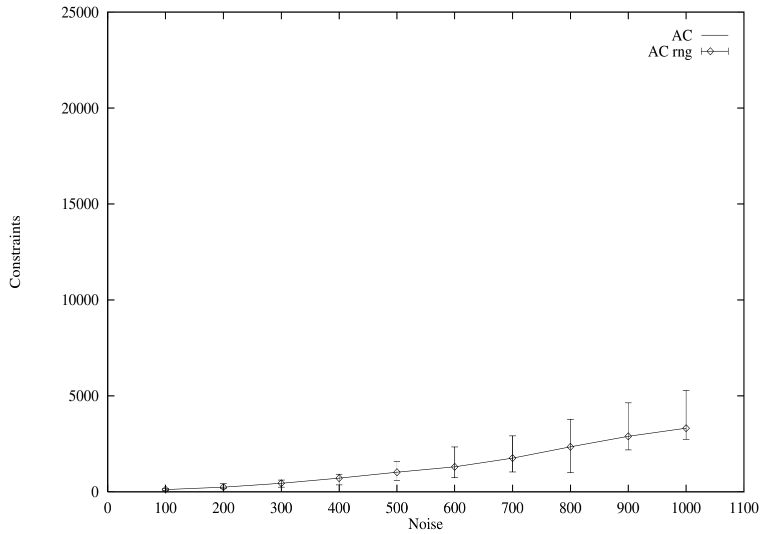

Figure 18: Arc Consistent (AC3) constraint checks.

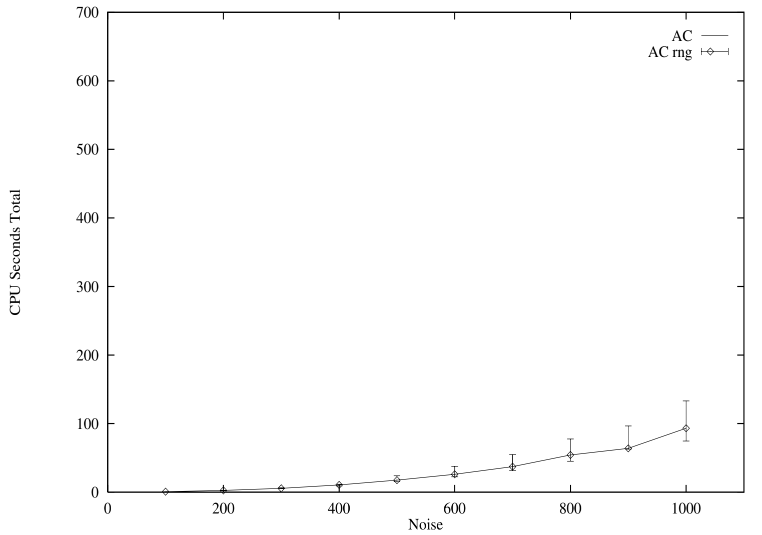

Figure 19: Arc Consistent (AC3) CPU seconds (total).

5 Summary and future directions

We have seen that it is feasible to recognize ADT instances in source code examples, and further that this recognition approach functions reasonably well even as noise increases. This result is a preliminary effort that is meant to help demonstrate the feasibility of implementing similar recognition as a portion of a program understanding methodology entirely based in the constraint satisfaction paradigm. The next phase of this work will see the verification that this result extend to the more complicated source code template recognition described in Figure 7 and Figure 6. Once this is completed, we will integrate this recognition in the larger framework of program understanding CSP search. This recognition is analagous to "source scanning" experts undertake so as to form early "hints" as to the context of a piece of source code to assist in the reduction of possible explanations of the remainder of the source. We intend to demonstrate the effectiveness of such an approach when combined with a systematic improvement of the Quilici model of program understanding that can exploit additional program structure during search.

Acknowledgments

This research has been carried out with the support of the National Science and Engineering Research Council, and the University of Waterloo Department of Computer Science. I would like to acknowledge the ongoing input of my supervisor Qiang Yang to many aspects of this work.

References

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.

Calingaert, P. (1988), *Program Translation Fundamentals*, Computer software engineering series, Computer Science Press, Rockville, Maryland.

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.

Ching, J. (1993), Computer imagery: An attributed hypergraph framework for visual cognition, *in* P. Thagard, ed., 'CS786G Course Notes', University of Waterloo. Ph.D. thesis proposal, pp. 14-46, Permission to copy granted by John Ching.

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

Dechter, A. & Dechter, R. (1987), Removing redundancies in constraint networks, *in* 'Proceedings of the 6th AAAI', pp. 105–109.

Dechter, R. (1992), 'From local to global consistency', *Artificial Intelligence* **55**, 87–107.

Dechter, R. & Meiri, I. (1988?), 'Experimental evaluation of preprocessing techniques in constraint satisfaction problems', *unknown* pp. 271–277.

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

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.

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

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 proble', *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.

Montanari, U. (1974), 'Networks of constraints: fundamental properties and applications to picture processing', *Information Sciences* **7**, 95–132.

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

Prosser, P. (1993), 'Hybrid algorithms for the constraint satisfaction problem', *Computational Intelligence* **9**(3), 268–299.

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

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

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

van Run, P. (1994), Domain independent heuristics in hybrid algorithms for CSPs, Master's thesis, University of Waterloo.

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* 'Proceedings of the 1993 DND Workshop on Advanced Technologies in Knowledge Based Systems and Robotics'.

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

Yang, Q., Tenenberg, J. & Woods, S. (1992), Abstraction in nonlinear planning, Technical Report cs-91-65 (ii), University of Waterloo.

Yang, Q., Tenenberg, J. & Woods, S. (1994), On the implementation and evaluation of Abtweak, Submitted for publication.

A The Quilici model of program understanding

The capturing of program knowledge is performed in two general directions: *top-down* and *bottom-up*. Top-down understanding reasons from possible program goals and attempts to generate possible plans for achieving the high level goals and then find connections between these and the source. Bottom-up understanding attempts to determine which programming plans might have any of the instructions themselves as components, and then iteratively attempts to infer successively higher level goals from these plans until the programmer's actual goals are identified or no further possible candidate plans exist to be matched against the source.

In the specific program understanding model described in (Quilici 1994), a strategy is outlined for recognizing programming plans, in which it is assumed that the input of the source, in this case Ccode, is in the form of an abstract syntax tree, and programming plans expressed hierarchically in a library. The intent is to translate to $C++$. and consequently recognize plans that can be mapped later to $C++$objects. The recognition process is outlined in Figure 20.

The essence of this approach is to select a bit of the source, try to fit it into an overall interpretation of the program according to known possible plan structure hierarchy, and repeat until all bits have been examined. Quilici exploits two other significant heuristics which help to sort through the many diverse possible explanations of each source bit: *specialization tests* and *index tests*. Specialization constraints or tests are defined for each plan and specify *how* it is possible to specialize a particular plan. For instance, a specialization constraint might say that some **Read-loop** plan which loops over a set of input *specializes* to a **Read-integer-loop** if the value being read was constrained to be an integer. In this manner, whenever a plan is recognized, the attempt is made to specialize it as much is possible based on current knowledge. Index tests act similarly, except that whereas specialization constraints say when one should attempt to specialize to a particular plan, index constraints say when one should attempt to match a particular *typically related* plan to the source. In this instance, a particular plan template is selected, and instances of this template are found in the source. These instances are then "fit" as possible candidate explanations for the corresponding plan component. For instance, if some plan component is matched successfully to an **Open-file-for-read** plan, and this plan *indexes* a **Read-element-from-file** plan, a search is made for any possible matches to **Read-element-from-file** in the source. A component that is matched to this plan is then instantiated. This is one example of a control heuristic that exploits the specific knowledge contained in the plan library to control search.

This algorithm describes a general approach to contextualizing components of the source one at a time. Context is built up according to a repetitive loop of identify, specialize and index. No concrete implementational examples are given, nor are the details sufficient to construct an implementation. In order to describe a potential search space it is going to be necessary to make several assumptions about the matching process itself, and the way in which matches are integrated into the plan library. If we view the construction of the explanation knowledge hierarchy as the search, then a search can be envisaged through a *space* of possible positionings of plan instances within the hierarchy, in which each node in the space is a complete assignment of each seen plan to a position in the hierarchy. Search commences from an initial point where no plan components are assigned to an "explanation" plan in the plan library, and as each component is *selected*, some instantiation is made to an explanation. Essentially, each component corresponds to a variable with a domain ranging over all plans in the plan library. In addition, the domain values are arranged hierarchically, with each plan potentially specializable in specific ways.

We now attempt to describe the behaviour of the Quilici approach in the following section, in an effort to describe more accurately the space of hypothesized explanations of source blocks or components. The search progresses by associating some source block B_1 with a plan P_1 from the hierarchy, specializing this $B_1 \rightarrow P_1$ relation as much as possible, indexing related plans and prompting an attempt or preference to instantiate other blocks B_i to these indexed plans. This process continues until all blocks B_i of the source code have assigned to a plan in the hierarchy. Since a $B_i \rightarrow P_j$ relation may be specialized according to the hierarchical knowledge structure, a plan assignment to a specializable block (variable) may be viewed as a narrowing of the range of possible values rather than an assignment in the traditional sense. This type of narrowing can be seen as similar to a constraint propagation (Dechter & Dechter 1987, Dechter & Meiri 1988?, Dechter 1992) application in which the reduction of one variable domain is the basis for testing and possible reduction of other variable domains. The search space that is being constructed consists of states which each contain a set of block-variables corresponding to the source components, and each variable is associated with a domain of possible plan instantiations, where each plan in the domain potentially represents a set of possible plan specializations. The search through this space is dictated by the particular chosen reduction of variable domains amongst all possible variable reductions. A solution state consists of one consistent plan assignment to each variable such that no plan may be specialized further based on existing source information. Traversal through the space of these states is made through the following transitions: *selection of a variable*, *assignment* of a plan or plan set to a variable, *specialization* of a variable plan assignment, and *indexing and locating* "likely" plans on the basis of local contexts which results in essentially an instantiation of some some component variable. If there is no allowable plan explanation for some component part B_i then a dead-end is encountered and any solution strategy would have to backtrack, essentially undoing a previous $B_i \rightarrow P_j$ assignment or reduction and systematically exploring the remaining options.

```
/* Initialize */
1. SELECT a component block B from component set L

/* Try to go from general to specific */
2. SPECIALIZE B as much as possible, ie
   Try all possible specializations Bx of B, and:
       on success:  add Bx to Recognized-Set, repeat (2) for Bx.
       on failure:  reject Bx specialization.

/* Check related plans */
3. Check if B INDEXES any other plans Ix in library,
   Try all possible index tests Tx for plans Ix specified for B,

/* Existence of Ix possible, no evidence yet */
If Tx fails ** missing part of Ix, place Ix on Indexed-Set

/* Existence of Ix confirmed, match incomplete yet */
If Tx succeeds && missing part of Ix, place Ix on Hypothesized-Set

/* Existence of Ix confirmed and mapped to source ok */
If Tx succeeds && match parts of Ix ok, add Ix to Recognized-Set

/* Existence of Ix refuted */
Reject Ix

/* Enlarge scope, perhaps something expected has been found */
4. Check index tests Iy for all plans Iy on Indexed-Set, Hypothesized-Set
on success, add Iy to Recognized-set
    on definite failure, discard Iy
on possible failure, retain  Iy

/* Loop til done */
5. Repeat 1 until component set L empty
```

Figure 20: Outline of Quilici understanding algorithm.

B CSP solution strategies

One extremely simple solution strategy for CSPs is to generate all possible permutations of variable domain value combinations and test a complete assignment to see if the problem constraints are met. Of course, such a method is very inefficient and will not be adequate for more complex examples involving more variables, larger domain sizes, and more elaborate constraint networks.

There are several approaches to solving CSPs which may be considered separately, or as hybrids. One class of these approaches is constituted by search-based strategies coupled with various heuristics which have been shown to reduce the overall search for certain types of problems. Another class of approach are the constraint-propagation methods, where the domain constraints are "propagated" between variables and domains, iteratively reducing the size of variable domains. This propagation may continue until all remaining variable domain values are "consistent" with one another and thus the problem is solved and all remaining assignment combinations constitute solutions, or propagation may continue until either some degree of consistency is achieved or variable domains are reduced to some point, at which time a reduced problem can be solved using a search strategy. We shall introduce search, propagation, and hybrid approaches briefly.

Propagation strategies for consistency

There exist a range of constraint propagation algorithms which result in varying amounts of *consistency* in the remaining CSP representation or graph. *Node consistency* involves removing all domain values for each variable such that a domain value violates a reflexive constraint. Node consistency is often referred to as 1-consistency. *Arc consistency* or 2-consistency works on constraints between variables, ensuring that each domain value for one of the variables has at least one possible "dual" in the other variable such that the inter-node constraint is satisfied. AC-3 (Mackworth 1977) is a well known algorithm that makes a graph 2-consistent, and subsequent extensions AC-4 (Mohr & Henderson 1986) and AC-5 (Van Hentenryck, Deville & Teng 1992) to this algorithm have appeared. Extending 2-consistency to graph path lengths of 3 is known as *Path consistency*(Montanari 1974, Mackworth 1977), and general consistency or n-consistency[16] algorithms have been presented (Freuder 1982, Cooper 1989) which further extend the consistent path-length in a graph.

For some problems it has been observed (Nadel 1989, Dechter & Meiri 1988?) that a preprocessing phase accomplishing node consistency for a CSP has been cost effective in terms of overall computational savings for finding ALL solutions in some domains, but also that the cost of achieving arc consistency has been found to be not cost effective. As a result of this work, efforts have been made in (Nadel 1989) to see if the actual optimal benefit point for consistency actually falls somewhere between level 1 (node) and level 2 (arc) consistency. The result has been several algorithms which achieve j-consistency (partial arc consistency),

[16] Note that a problem graph of n variables that is made n-consistent is solved without search.

where j is between 1 and 2. Results in some experimental domains have shown that these partial arc consistency algorithms combined with search outperform those utilizing only node consistency as a pre-search step (Nadel 1989).

Search strategies and hybrids

CSPs can be solved using search alone, using consistency propagation alone, with search and using consistency propagation as a preprocessing phase, or by using some hybrid approach interleaving constraint propagation and search. As we have stated, the CSP search space consists of nodes where each node consists of a set of particular instantiations of variables with domain values. The initial or start node represents the least constrained version of the problem. From this start node it is possible to (1) select a variable V to instantiate (the variable order), (2) select a domain value to instantiate variable V with (the domain order), or (3) select some subset of the graph to make *more* consistent via propagation. Using these transitions the entire possible space of the problem may be searched. There are a finite number of variables, n in our problem. The search space therefore has only n instantiations possible. There are only a finite number of domain values, m, for each variable. Each variable can only be instantiated once. A constraint propagation stage either (a) results in a domain reduction for some subset of variables, or (b) results in no reduction at all. Consequently, if we regard the search space as containing one levels for each instantiation of a variable, and having maximum width of m for each variable, then it is clear that the all "solution" nodes will be encountered at the leaves of the tree at a "depth" of n levels. We can see therefore that any depth-first or breadth-first search will be complete in the sense that any or all existing solution(s) will be found, eventually. In a search for one solution, or for one "good" solution it will not be necessary to examine the entire space, and therefore search control and "good" selection and in-search planning of variable selection can dramatically affect performance.

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*).

Standard algorithms for solving CSPs include *Simple Backtracking*, *Forward Checking*, and *Partial Lookahead*, *Full Lookahead*, and *Really Full Lookahead* (Haralick & Elliott 1980). In (Nadel 1989), an empirical comparison of the performance of these algorithms is described within a unified interpretation framework in which these algorithms are described primarily in terms of the degrees of arc consistency which is performed at the nodes of the search tree. Essentially each algorithm is depicted as a combination of "pure" or simple tree search and some limited attempt at propagating consistency between variable instantiations.