Program Understanding : A Constraint Satisfaction Modeling Framework; Understanding as Plan Recognition

Steven Woods [1]
University of Waterloo
Canada

Alex Quilici [2]
University of Hawaii
U.S.A.

Qiang Yang [3]
Simon Fraser University
Canada

[1]Computer Science Department, Waterloo, Ont., Canada, N2L 3G1. sgwoods@logos.uwaterloo.ca

[2]Department of Electrical Engineering, Holmes Building 483, Honolulu, Hawaii 96822. alex@wiliki.eng.hawaii.edu

[3]School of Computing Science, Intelligent Software Systems Lab (ISG), Burnaby, British Columbia V5A 1S6. qyang@cs.sfu.ca

Abstract

Different program understanding algorithms often use different representational frameworks and take advantage of numerous heuristic tricks. This situation makes it is difficult to compare these approaches and their performance. This paper addresses this problem by proposing constraint satisfaction as a general framework for describing program understanding algorithms, demonstrating how to tranform a relatively complex existing program understanding algorithm into an instance of a constraint satisfaction problem, and showing how this facilitates better understanding of its performance.

Plan recognition is the task of interpreting the actions of agents in the environment, in the context of the knowledge we possess about how action occurs in the world, and why. The recognition task involves constructing a mapping, possibly partial, between an existing repository of plan and domain knowledge and a set of dynamic observations of a subset of the actions taken toward a goal. Program understanding can be viewed as a special case of plan recognition, where the task is to recognize the plans programmers have used in constructing a particular piece of legacy source code. However, program understanding differs from generalized plan recognition in that a complete set of action observations is the basis of goal determination. This paper discusses, in detail, how this difference leads to inadequacies in applying typical plan recognition algorithms to program understanding. Program understanding can instead be viewed as a special case of plan recognition which is particularly amenable to constraint satisfaction techniques.

Contents

1 A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms 4

1.1 An Existing Plan-Based Program Understanding Algorithm 5
1.2 Program Understanding as a CSP 9
1.3 DECODE's Understanding Algorithm as CSP 13
1.4 Experimental Discussion 16
1.5 Conclusion 17

2 Program Understanding and Plan Recognition : A Comparative Study 21

2.1 Introduction 21
2.2 The Plan Recognition Paradigm 22
2.3 Program Understanding Overview 26
2.4 Program Understanding as Plan Recognition 30
2.5 Conclusion 33

List of Figures

1.1 An example code pattern. 7
1.2 Our algorithm for automatically recognizing plan instances in code. 18
1.3 An example plan in the MAP-CSP representation. 19
1.4 MAP-CSP representation of code patterns. 19
1.5 Forward Checking, DR (95% conf. interval). 20

2.1 Action hierarchy for the cooking domain. 23
2.2 Conceptualizing source with a plan library. 27
2.3 Example program plan hierarchy. 31

Chapter 1

A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms

Introduction

Over the past decade, researchers have proposed and implemented a wide variety of plan-based program understanding algorithms (Quilici 1994, Kozaczynski & Ning 1994, Kozaczynski & Ning 1992, Wills 1992, Wills 1990, Hartman 1991, Johnson 1986). While some of these research efforts have presented promising empirical results in mapping plan libraries to reasonably sized (up to 1000 lines) legacy source code(Wills 1992, Quilici & Chin 1995, Woods & Yang 1995*b*), none have been clearly demonstrated—either analytically or empirically—as scaling up for use in understanding real-world legacy systems. In addition, little work has been done in comparing the relative performance of these approaches or analyzing in detail the similarities and differences between these algorithms. In part, this situation has resulted because the algorithms tend to be based upon different representational frameworks (such as flowgraphs, components and constraints, regular expressions and transformation rules, and so on) and to use collections of heuristic tricks to improve performance (indexing, specialized rule and constraint ordering, and so on).

As a consequence, it is difficult to systematically compare these different approaches or to understand how their performance will be affected by variants in the plan library (such as adding large numbers of new plans) or programs being understood (such as changing the distribution of basic syntax tree items and the dependency relationships between them)

What is needed is a framework for describing these algorithms that allows ready empirical and analytical comparisons of their behavior. In earlier work (Woods & Yang 1995*b*, Woods & Yang 1995*a*), Woods and Yang demonstrate how a particular approach to program understanding can be viewed as a *constraint satisfaction* problem (CSP)[1]. It is therefore natural to wonder whether other, existing program understanding algorithms, despite their differing representations and heuristic tricks, can also be mapped into this constraint satisfaction framework. If this framework is sufficiently general to unify these approaches, then we can take advantage of it to compare their relative performance and better understand where these algorithms succeed and fail in attacking the program understanding problem. In addition, we can potentially achieve improved scalability of these approaches by augmenting them with the mechanisms developed for efficient heuristic solving of different classes of constraint satisfaction problems. These mechanisms range from ranging from global(Kondrak & van Beek 1995) and local search-based methods(Sosic & Gu 1990, Minton, Johnston, Philips & Laird 1992, Yang & Fong 1992), constraint-propagation problem simplifications(Nadel 1989, Dechter 1992, Prosser 1993), hierarchical exploitation of problem structure(Freuder & Wallace 1992), as well as hybrid combinations of these approaches.

This paper demonstrates how one well-known heuristic algorithm for program understanding can be placed within a constraint satisfaction framework, how this improves our understanding of its performance, and shows how this viewpoint facilitates comparing its performance with other program understanding algorithms. Section 1 describes this algorithm. Section 2 provides an overview of how program understanding can be viewed as a constraint satisfaction problem. Section 3 shows how an existing algorithm can be turned into a constraint satisfaction problem, while preserving both its representational framework and heuristic tricks. Section 4 discusses the performance of a key aspect of this constraint satisfaction approach. Section 5 describes our future research path and our conclusions from our current work.

1.1 An Existing Plan-Based Program Understanding Algorithm

This section describes an existing plan-based program understanding algorithm (Quilici 1994). This algorithm was derived from studies of users doing bottom-up understanding on functions in C code (Quilici 1993) and used in a cooperative program understanding environment (Quilici & Chin 1995). As a result, it has a variety of heuristic tricks to make it more efficient to help it model the behavior of these users and to ease the effort required in providing program plans.

This algorithm's representation of a program plan was based on the Concept Recognizer (Kozaczynski & Ning 1994, Kozaczynski & Ning 1992). The Concept Recognizer divides plans into two parts: a description of the plan's attributes (which are instantiated when a plan instance is recognized) and a set of common implementation patterns. It represents these code patterns as a combination of *components* (the particular language items or subplans that

[1]See (Kumar 1992) for an overview of constraint satisfaction.

must be recognized to have a potential instance of the plan) and *constraints* (the relationships that must hold between these components).

Given this representation, the Concept Recognizer takes a library-driven approach to recognize plans. It takes each code pattern in a plan library, matches its components against the program, and then applies constraints to the set of candidate plans (actually, it tries to interleave constraint checking and matching). When a component can itself be a plan, the algorithm recursively tries to recognize instances of that plan.

The Concept Recognizer's representation of plans is both simple and clear and the algorithm is successful at recognizing plans in real-world COBOL programs. However, the algorithm is slow and does not scale well, either with program size or plan library size (Kozaczynski & Ning 1994). DECODE's program understanding algorithm tried to address these problems in two ways. First, it is code-driven (bottom-up) rather than library-driven (top-down). While library-driven approaches consider all plans in the library, code-driven approaches consider only the subset of those plans that contain already-recognized components. Second, it relies on an extended plan representation that supports careful indexing and organization of the plan library to reduce the number of constraints that must be evaluated and the amount of matching that must take place between the code and the plan library.

Representation

Figure 1.1 contains several examples of DECODE's extended plan representation. As in the Concept Recognizer, each plan consists of a set of components and constraints. For example, one implementation of the plan `TRAVERSE-STRING` (which captures the common notion of traversing each character in a C string) consists of a set of components: a `DECL-ARRAY` to declare the character array, a `ZERO` sub-plan to initialize the index variable to zero, a `LOOP`, two `ACCESS`es to access an indexed element (one for a comparison, the other to use the array element), a `BIN-OP` to compare the indexed element with a null character, and an `INCREMENT` to update the index variable. However, not any combination of these components is an instance of the plan. There must also be a variety of data and control dependencies between its components, such as a data dependency between the test of the index variable and its initialization. Only if all these constraints hold do we have an instance of the plan `TRAVERSE-STRING`.

In addition to the basic components and constraints, each plan has an index that says when it should be considered (that is, fully matched against known program pieces and recognized plans). The index combines a plan component with one or more plan constraints and suggests that the plan should be considered whenever this component is encountered and the specified constraints hold. `TRAVERSE-STRING`, for example, is indexed by an `ACCESS` that is contained within a `LOOP`. That means the understander considers this plan each time it encounters an `ACCESS`, not every time it encounters any `INCREMENT`, `ZERO`, `BIN-OP`, `LOOP`, or `DCL-ARRAY` (as in most bottom-up approaches). Evaluating the index involves checking

```
define TRAVERSE-STRING(String) isa TRAVERSE-PLAN
define PRINT-STRING(String) isa PRINT-PLAN
define PRINT-CHAR(Char) isa PRINT-PLAN
define ZERO(Dest) isa ASSIGN-PLAN

plan TRAVERSE-STRING(String: ?a)
  components
    decl:     DECL-ARRAY(Name: ?s, Items: ?max, Type: char)
    init:     ZERO(Dest: ?i)
    loop:     LOOP(Test-Result: ?r, Body: ?body)
    access1:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val1)
    test:     BIN-OP(Op1: ?i, Op2: ?val1, Op: !=, Res: ?r)
    access2:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val2)
    update:   INCREMENT(Op: ?i)
  constraints
    declbef:  ControlPath(decl, loop)
    initbef:  DataDep(test, init, ?i)
    acc1bef:  DataDep(test, access1, ?val1)
    testin:   DataDep(loop, test, ?r)
    acc2in:   ControlDep(access2, ?body)
    updaft:   DataDep(access2, update, ?i)
  index
    access2 WHEN accin

  implies PRINT-STRING(Srting: ?a)
    with
      dump:     PRINT-CHAR(Source: ?value)
    when
      dumpaft:  DataDep(dump, access2, ?v)

plan PRINT-CHAR(Char: ?c)
  specializes Call-Function(Name: putchar, Args: ?c)

plan ZERO(Item: ?i)
  specializes Assign(Dest: ?i, Value: 0)
```

Figure 1.1: An example code pattern.

whether its indexing constraints hold (which may in turn involve trying to match additional plan components). In this case, it involves determining whether the `ACCESS` is contained within the body of a `LOOP`.

The idea is that indexes suggest when plans are *likely* to occur as opposed to when plans *might* occur. This has the potential to cut down on the number of plans in the library that are considered during understanding, as any plan that is not indexed by the elements of a given program will never be considered. It also has the potential to significantly reduce the number of times any given plan is considered by a bottom-up understander from the total number of times any of its components occur in the program to the number of times its indexing component occurs in the program. Finally, it has the potential to reduce the amount of matching and constraint evaluation that takes place while recognizing instances of a particular plan. Ideally, the recognition process should always evaluate any constraint that will fail as soon as possible, since a single failed constraint eliminates a plan instance from further consideration, whereas all constraints must succeed before a plan can be recognized. Because indexing places a partial ordering on both matching (with the indexed component of the plan bound first) and constraint evaluation (with the indexing constraints evaluated first), the better the indexing constraints are as a predictor of a plan's presence, the fewer uneeded constraints will have to be evaluated.

In addition to indexes, our representation extends the Concept Recognizer to allow plans to be defined as being conditionally implied by other plans. After the understander recognizes a plan that conditionally implies another plan, it checks whether these conditions hold (which involves checking for additional components and evaluating additional constraints). For example, the plan `TRAVERSE-STRING` implies the existence of the plan `PRINT-STRING` when there exists an additional `PRINT-CHAR` that is conceptually contained within the `LOOP`.

The idea behind implications is to take advantage of small differences between the implementations of related plans, so that one plan can be recognized as a slight modification or extension to another. Essentially, plan implementations are organized in a discrimination net, which allows the understander to use indexing to retrieve general plans to try first and then to use small, additional incremental tests to recognize more specific plans.

There are two alternatives to implications. One is to have related plans be complete, stand-alone implementations that individually contain all necessary components and constraints. `PRINT-STRING`, for example, could be defined so that it contains all of `TRAVERSE-STRING`'s components and constraints. This approach, however, leads to duplicate component matching and constraint evaluation that can be eliminated by explicit implication links. The other alternative is to have the specific plans contain the general plans as elements. `PRINT-STRING` could be defined to contain `READ-ALL-RECORDS` as one of its components and to have additional constraints that relate it to their other components. The problem with this approach is that the additional constraints may require access to `TRAVERSE-STRING`'s implementation (such as a control flow relationship involving its `LOOP`), which then forces `PRINT-STRING` to have additional implementation-oriented attributes. Although this is just as efficient as implication links, it makes the definitions of plans much more difficult. So implications allow a natural representation of relationships between plans without adding a significant cost.

Finally, our representation allows plans to be defined as *specializations* of other plans; that is, as a set of constraints on an existing plan's attributes. For example, the plan `ZERO` is defined as a specialization of an `ASSIGN` whose Source is `0`. These specializations correspond to plans that contain a single component (the plan being specialized), that are indexed by that component, and that have constraints on that component's attributes. In fact, at definition time, these specializations are automatically translated into standard plan definitions.

The idea behind specializations is to make it easy to define one common class of plans and to encourage the definition and use of specialized plans as components and indexes. This simplifies the definition of higher-level plans that contain specialized plans as components by reducing the number of constraints that must be specified. This ability is simply a convenience, however, with no performance implications.

Control

Figure 1.2 shows the actual algorithm used by our original program understander. The basic idea is straightforward: run through the program tree and, whenever a component is an index for a plan and its indexing constraints succeed, match the remaining pieces of that plan against the code and evaluate the constraints on the partial plan instances formed by the matching process. In addition, whenever a plan is recognized and implies another plan, attempt to match the additional components and evaluate the additional constraints. Then for each plan recognized, recursively see if it indexes any plans.

There are several complications. One is that at the time an index is evaluated, components that are themselves plans may not have been recognized yet. For example, the `INCREMENT` in `TRAVERSE-STRING` may be a subplan that is recognized after the index triggers consideration of `TRAVERSE-STRING`. To avoid this problem, our algorithm assumes that the plan library is organized in layers, where each layer contains the plans dependent only on items in the previous layer. At the bottom are plans like `PRINT-CHAR` and `INCREMENT` that depend only on abstract syntax tree items. At the next level are plans, like `TRAVERSE-STRING`, that depend on these subplans. The algorithm then breaks the indexing process up into layered traversals through the program tree, first seeing if anything in the first layer is indexed, then if anything in the next layer is indexed, and so on. Implications are handled in a similar way, with any plan implied by another plan placed in a layer that is both above it and above any of its new subcomponents.

The other complication is that evaluating constraints and binding components against the program tree must be interleaved. A simple approach to recognizing plans would form all the possible combinations constructed by binding each of its components against program tree entries and then evaluate the constraints on these components. However, that is far too inefficient. Our alternative is to have an ordering for constraints and to form combinations only as they become necessary to evaluate these constraints.

1.2 Program Understanding as a CSP

The algorithm described in the previous section has a number of nice properties, such as limiting the number of plans considered, components matched, and constraints evaluated, as well as modeling an empirical study of programmers understanding code. However, it also has some drawbacks. It is relatively difficult to understand and analyze and to compare in detail against other program understanding algorithms. To remedy these drawbacks, this section provides an overview of how program understanding algorithms can be viewed as a constraint satisfaction problem, and the next section will demonstrate how the preceding algorithm can be placed in that framework.

Constraint Satisfaction Problems (CSPs) consist of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints among the variables which restrict domain value assignments. A solution to a CSP is a set of domain value-to-variable assignments such that all inter-variable constraints are satisfied.

PU-CSP

Program understanding can be represented as a constraint satisfaction problem, called PU-CSP, in the following way (as we demonstrated in earlier work (Woods & Yang 1995*b*)).

We assume the source code is divided into a variety of *blocks*. A block can be anything from a single statement to a program slice or other arbitrary collection of related statements. The program understanding problem is then to explain what the entire program does by explaining what each block does and then determining what various sets of blocks do in conjuction. The possible explanations correspond to a set of plans in a hierarchically organized program plan library, and an explanation of the source program is a mapping from members of this library to the program's components. The PU-CSP problem is to determine this set of possible explanations for a given set of program blocks using a constraint satisfaction approach.

The variables in a PU-CSP are the blocks in the program to be understood. The domain for each variable ranges over all of the plans that could possibly explain that block. However, this is only a subset of the plan library. The block may be of a particular type, in which case only plans that contain that type as a component can explain it (such as when the block corresponds to a single action in the AST), or it may have particular input and output types that are matched by only a small set of plans (such as when the block represents a function).

The constraints fall into two categories: *structural* constraints between blocks and *knowledge* constraints between plans. The structural constraints correspond to structural relationships between blocks (e.g., data-flow, control-flow, and temporal-ordering). The knowledge constraints correspond to restrictions on the ways plan may be connected (e.g., that a plan must fall into a particular category, that a plan must have certain components, those components have a characteristic flow of data between them, and so on). A mapping between the plan hierarchy and the blocks is a possible explanation only if the set of knowledge constraints is consistent with the set of structural relationships present in the source code.

PU-CSP is seeking a global explanation of all or part of a program's source based upon its particular components and their structural relationships. However, program plans in the plan library may be based upon sub-plans at lower levels of abstraction. In addition, programmers often take advantage of their recognizing familiar functionality by using these partial explanations when explaining large blocks or chunks of code(von Mayhrhauser & Vans 1995). We can therefore improve on PU-CSP by augmenting it with a mechanism to locate the initial set of possible, low-level explanations for various blocks. This mechanism is handled by a separate constraint satisfaction problem, called MAP-CSP.

MAP-CSP

MAP-CSP represents the problem of locating all instances of a program plan template in the source code (i.e., mapping this plan to directly to source code entities). The variables in the MAP-CSP are the components of the plan. The domain for each variable ranges over source code components of compatible types, and the actual occurrences of each of those components in the source code correspond to possible domain values for the variables. The components within a given plan are constrained by various data-flow and control-flow relationships that must hold between them, which are represented as inter-variable constraints in the MAP-CSP. A solution to the MAP-CSP problem is therefore any assignment of domain values (AST elements) to template variable (program plan parts) that satisfies the constraints among the variables (data-flow and control-flow relationships). A solution is an instance of the program plan template which we have identified in the source code, and thus explains that part of the source code being mapped.

Given a plan library, repeated application of MAP-CSP can be used to recognize all instances of plans whose components correspond solely to abstract-syntax tree elements.

The essence of the PU-CSP/MAP-CSP approach is that PU-CSP attempts to combine individual MAP-CSP solutions that represent only some subset of all program plans in the hierarchy. The plan instances identified with these MAP-CSP solutions are integrated into a partial explanation covering some number of source code components which may be thought of as blocks of "locally explained" source code. Thus, at any point in time there is some set of blocks "explained" and some set "unexplained", with these blocks related structurally through data and control flow relationships.

Similarly, the explained blocks are known to relate in specific ways to other program plans in the hierarchy. For instance, consider the case where three blocks A, B and C exist such that control or data flow constraints exist between them. Suppose blocks A and B have been mapped with MAP-CSP to particular program plans in the library, A1 and B1 respectively. Block C possibly corresponds to any of 3 different program plans in the hierarchy: C1, C2 or C3. The knowledge constraints present in the library for program plans A1 and A2 may now be usable to constrain the range of block C. For instance, if A1 is known to precede C2 according to the library but it is the case that program block A is structurally constrained in the source to follow block C2, then C2 can be safely eliminated as a possible explanation of C. This process is simply an application of knowledge constraints against structural relationships, and corresponds to a limited form of constraint propagation. This behaviour could also be though of as search in which the leaf node representing A=A1, B=B1, and C=C2 is pruned or rejected as a potential solution.

An Alternative CSP-Based Approach

Another way to view the program understanding problem is as an ordered set of plan matches in the flavor of MAP-CSP. If the plan library is constructed in layers, so that plans at each level are built only from plans at lower levels, it would be possible to use MAP-CSP to find all instances of the plans based only on items in the AST, and then continue up the hierarchy, matching each successively higher level. This possibility gives rise to the question: Why bother with PU-CSP?

The problem with a strictly bottom-up application of MAP-CSP is that it relies on a mapping of every plan instance in the library. As a result, many independent MAP-CSPs must be solved, in the sense that it is not clear how in general solving one MAP-CSP can be exploited to reduce the effort made by other MAP-CSPs. (It is possible for MAP-CSPs at one level to contribute to the solving of MAP-CSPs at a higher-level in that failing to recognize certain plans in one MAP-CSP quickly eliminates the consideration of the higher-level MAP-CSPs involving those plans. However, what is not clear is how MAP-CSPs at one level can contribute to other MAP-CSPs at the same level.) In contrast, if we consider the PU-CSP approach as a global strategy for controlling the application of MAP-CSPs and for integrating the MAP-CSP solutions for local code portions, it may be possible to restrict the range of possible explanations for larger code components more effectively.

In any case, a purely layered approach is not entirely satisfactory when we consider real-world use of program understanding tools. In particular, any real-world program understanding tool is going to involve some interaction with users, as there is always going to be some idiosyncratic code that doesn't correspond to any plan in the existing plan library (Quilici & Chin 1995). As a result, the program understanding task corresponds to efficiently partially reverse-engineering the code. In the repeated application of MAP-CSPs, it's difficult to imagine how the programmer can help the process. However, in the PU-CSP approach, both the algorithm and the programmer can exploit local partial solutions to restrict other, possibly higher-level solutions. Larger code components such as procedures or functions form nicely coupled code chunks with clearly defined constraint relations among them in the form of calling and type relationships. The identification of plans that interact with one of these function blocks can potentially reduce the combinations of explaining a set of these function blocks.

Finally, earlier work with spatial templates (Woods 1993) has demonstrated that sets of complex constraints, such as those involved in MAP-CSP's plan templates, are very difficult for experts to quickly identify in noisy situations, such as is provided by confusing or cluttered source code. Interative large-scale understanding of complex spatial situations was greatly assisted by local identification of difficult-to-see spatial relationships. The idea in this earlier work was that these micro-solutions can be thought of as initial building blocks on which to build expert-level explanations. Essentially, applying this idea to program understanding suggests doing as many of these micro-observations (MAP-CSPs) as is computationally affordable and then attempt to couple those with the macro constraints of the larger PU-CSP so as to maximize the effectiveness of the high-level easy to identify constraints such as inter-function control and data flow.

Another alternative approach is to carefully interleave low-level and high-level MAP-CSPs. For example, one need not apply all the lowest MAP-CSPs first but rather apply the lowest ones in a particular portion of the planning hierarchy, and then higher ones atop these low ones, until the point at which a larger code block has been successfully explained. Then this larger context explanation could be used to select the next MAP-CSP to match, and so on. As a result, this interleaving may be able to exploit some of the structued constraints that exist between high-level plans and source code. However, this is exactly what PU-CSP is meant to do.

1.3 DECODE's Understanding Algorithm as CSP

There are two primary concerns in modeling a particular program understanding methodology in a constraint-based framework: representation and control. We must ensure that the CSP representation is general enough to capture the complexities and nuances of the original while not abstracting away important details, and we must ensure that the original control strategy can be interpreted in terms of a particular control strategy for solving CSPs.

Representation

In the memory-based program understanding problem representation described earlier, there are two primary representational parts: the individual program plans, and the hierarchical plan library.

The individual program plans (as in Figure 1.1) are represented in terms of components and constraints. In our CSP representation (for MAP-CSP), we model each of these components as a variables. Each variable has a domain ranging over the actual statements in the program that satisfy a set of constraints on the "type" of the variable. These "type" constraints may be thought of as reflexive in that they affect one variable only. They are derived from the partial naming and typing information provided in the component description. For instance, `DECL-ARRAY` is given as an array declaration structure with 3 parameters: a name that locally is allowed to range over any value (unconstrained), the size of the array (also unconstrained), and a type of array element (constrained to character). Thus, `DECL-ARRAY` "matches" any program statement that declares an array (in any fashion) such that the declaration satisfies the constraint that it is of type character of any size or any name. It is easy to imagine components that would map into more tightly constrained CSP variables.

MAP-CSP models the memory-based constraints among program plan components as CSP constraints among variables. A direct mapping exists between the function of constraints in the memory-based approach and the CSP approach. In the example plan, a constraint `ControlPath` exists between the `DECL-ARRAY` and the `LOOP` such that the `DECL-ARRAY` logically precedes the `LOOP`. This is mapped to the CSP representation directly, where any instance of the variable corresponding to `DECL-ARRAY` is constrained to logically precede any instance of the variable corresponding to the `LOOP` component. Figure 1.3 details the variables and constraints of the resulting MAP-CSP for our example plan.

We have seen a direct mapping can be made between components and variables as well as between component constraints and CSP constraints. In particular, these mappings are exactly those required for the specification of the MAP-CSP sub-problem. However, it is also possible that, in the memory-based model, the individual components are subplans rather than elements of the AST.

We currently deal with hierarchical plan structure through a layered plan library and applications of MAP-CSP a layer at a time. The MAP-CSPs at each subsequent layer include all of the recognized plans at the previous levels as part of the domain of variables. We rely on indexing to guarantee that the MAP-CSPs in a given layer fail quickly if the indexed component hasn't been recognized from the previous layer. And we rely on the MAP-CSPs at the lower layers to locate the possible domain values for the components at the higher levels. This implies that PU-CSP is not strictly necessary for our representation of our memory-based recognition algorithm (although it's still useful as part of a general constraint-based framework).

The parts not yet mapped directly as constraints or components to the CSP methodology are the `INDEX` entries of a program plan, and the `IMPLICATION` entries. We also have not specified what is to be done with respect to actually matching the program plan to the source program. In the next subsection we discuss how these elements are combined as search control for MAP-CSP.

Control

Our memory-based program understanding algorithm traverses the program source[2] and tries to match a particular program plan whenever it encounters an *index* for that plan. Program plans are organized in layers, with indexed plans at the lowest level of the hierarchy matched first, with indexed or implied plans at higher abstraction levels matched subsequently. Thus, a pass of the source involves checking each statement against the list of indices for a possible match. A possible match triggers a closer inspection of the source for an instance of the matched program plan. This closer inspection is exactly an instance of MAP-CSP in which the index part of the program plan template has already been identified.

Essentially, the performed MAP-CSP utilizes a strict ordering in which the components and constraints in the plan's index are matched first, with a successful index signaling the requirement to continue searching further. If the rest of the program plan components and constraints are successfully matched to the source code, MAP-CSP has identified an instance of the plan. What has been created here is a view of the CSP in which a subset of the variables and constraints are solved first, and further, in a particular order. We may view this as a hierarchical view of the CSP in which the "key" portion is "more important" and thus matched first.[3] If this key portion of the template contains variables that match only a

[2]Actually, it traverse the abstract syntax tree.

[3]See (Freuder & Wallace 1992, Yang & Fong 1992) for a detailed discussion of hierarchical CSPs and their solutions.

small subset of all possible program components, and constraints that are restrictive then this may be seen as an attempt to order the constraints so as to reduce the branching factor and size of the subsequent search space. An index by definition is a signifier of uniqueness, and thus it is only sensible that an *index* is matched only infrequently. The result is that indices in memory-based understanding are interpreted as orderings on variables and constraints in MAP-CSP.

We handle implication in a similar way to indexing. Any plan that is implied by another can be thought of as being indexed by the plan and any of the implication constraints. As a result, when we process a layer of plan library, we also do MAP-CSPs for any plans in that layer that are implied by plans at earlier layers, with the domain variables of each MAP-CSP being set up based on the bindings from the previously recognized plan.

An Example of MAP-CSP In Action

We have implemented a MAP-CSP version of the memory-based algorithm. This new algorithm models the identification of program plan instances in the following way. A CSP is formed in terms of variables mapping from the program components of the program plan, reflexive variable constraints mapping from the type information of the program plan components, and inter-variable constraints mapping from the data flow and control flow relations in the program plan itself. Each variable ranges over some subset of the program's statements. Once the problem is formulated in this way, the index information specified in the memory-based model is used as a preliminary ordering heuristic for the constraint set.

Figure 1.4 is an example showing how the portion of the plan of Figure 1.1 corresponding to the index is actually represented.

The index is formed as an instance of a particular kind of array access which is determined to reside in a loop structure. We represent the array access (labelled `ACCESS` in Figure 1.1) as a variable `v3` of a particular type of assignment, `Assign`, for assigning a value to a character array. We map the complex operation `LOOP` in Figure 1.1 as a combination of a variable `v1` of type `While`, a variable `v2` of type `Begin`, and a variable `v3` of type `End`. We represent the program plan index constraint that the `Assign` exist inside the control environment of the `While` with the pair of precedence constraints placing `v3` after the `v2` instances and before the `v4` instances.

The control proceeds roughly as follows. The first variable, `v3`, is matched against all program statements, giving a domain ranging over all `Assign` candidates of the appropriate type. This range can be thought of as the branching factor of the top of the search space. A large range signifies a poor key choice. Now, the constraints are applied in index-order. All satisfying instances of `v1` are identified such that `v1` is before `v3`. Next, for each instance of `v1`, a corresponding `Begin` instance of `v2` is identified. The `End` instances of `v4` are now identified according to the naming identifier of the corresponding `Begin` instances `v2`. A solution is then found for each set of assignments of domain values to variables such that `v3` is before `v4`. Each solution is an instance of an index hit that is a candidate for further search to locate full plan instances. The additional components are given domain ranges and then the remaining constraints are applied.

A typical CSP strategy would attempt to order variables and constraints independent of the particular enforced ordering implied by the memory-based index. In particular, in many intelligent backtracking CSP solution schemes this process would be undertaken dynamically rather than statically, thus taking advantage of particular problem characteristics in reducing the search space rather than relying on a pre-determined belief about the nature of the source examples that will be encountered. We discuss a particular approach used for comparison purposes in the next section.

1.4 Experimental Discussion

Earlier work(Woods & Yang 1995*b*, Woods & Yang 1995*a*) showed that MAP-CSP problems with 5 components and 9 inter-component constraints could be solved with relative efficiency for significantly sized source code blocks up to about 500 lines of code. Subsequent experimentation with well-constrained program plan templates modeled after the `TRAVERSE-STRING` example of Figure 1.1 with 9 components and 20 constraints has shown even more promising results. Figure 1.5 outlines some preliminary results for this MAP-CSP in randomly generated program sources ranging from 50 to 1000 lines of code. We see that the results scale quite well over this range, with the time required for MAP-CSP to complete for 50 lines laying well below 1 second (average 250 constraint applications), and for 1000 lines of code approximately 1 minute[4](average 30,000 constraint applications). Results graphed are for 10 problem instances at each legacy source size interval.

The stability decreases with increasing problem size, however, even the worst case results are promising. This result is achieved with the relatively straightforward intelligent backtracking algorithms known as Forward Checking with Dynamic Rearrangement (FCDR) of selected variable during search based on smallest domain size. This approach may be thought of as a dynamic approximation of an index in the memory-based methodology. In the absence of a selected index (essentially undirected backtracking), results have a much lower stability and in fact problems of 400 lines of code require almost 40 times as many constraint checks as for the FCDR approach.

We have implemented the memory-based algorithm formulated as an instance of MAP-CSP with a particular indexing strategy and we shall present results from experiments with this strategy in future work. We expect to see a result which indicates that a well-chosen index results in highly efficient strategies, a poorly chosen index in inefficient strategies, however, an open question remains as to whether it is possible to either statically or dynamically determine a *better* index automatically. Since we view any index is as a particular constraint ordering, it is quite conceivable that the best index from the point of view of the

[4]Note that these results are obtained on a Sparc 10 workstation.

memory-based methodology can be approximated better for a particular problem instance than in general.

1.5 Conclusion

The constraint-based approach has several clear advantages over previous methodologies. The first is its *generality*: we have demonstrated how one earlier, complex algorithm can be represented as a CSP with a particular representation and control strategy. This gives us hope that we will be able to do the same for other program understanding algorithms as well. In fact, we are now in the process of doing this same task for other published program understanding algorithms. The result should be a deeped understanding of the commonalities and differences of these algorithms.

Another key advantage is an increased ability to address *heuristic adequacy*, or *scalability*. By casting program understanding as a CSP, the previously known constraint propagation and search algorithms can potentially be adapted to improve these algorithms. In addition, we can compare the efficacy of specific heuristic tricks such as indexing to different methods of solving constraint satisfaction problems. It may well prove that existing methods are sufficient to achieve indexing's performance without the need to index, or alternatively, that we will see exactly what benefits are provided by the specific knowledge used in indexing (such as the likelihood of certain components indicating the presence of certain plans or the relative cost of evaluating various constraints) over heuristic constraint propagation methods.

The final advantage is that it becomes possible to complete a systematic study of different search heuristics, including both top-down and bottom-up as well as many other hybrids (such as the comparing the layered MAP-CSP approach to the mixed PU-CSP/MAP-CSP approach) in order to determine which ones perform the best on understanding source code.

Although we have just begun studying program understanding algorithms in terms of this constraint satisfaction approach, it appears very promising as a unifying framework for describing and comparing program understanding algorithms. Our hope is that it will lead us to a deeper understanding of existing program understanding algorithms and ultimately to a program understanding approach that scales.

```
Plan Recognition Algorithm

  • Initialize the program tree ($PT$) to the set of elements in the program's abstract syntax tree
  • For each plan library layer $L$:
      – For each element $E_i$ in $PT$:
          ∗ For each plan implementation $P_j$ in $L$ indexed by $E_i$:
              · Form the set of partial plan instances ($PPI$) that result from binding $E_i$ to
                each $P_j$.
              · Replace $PPI$ with the set that results from processing the indexing constraints
                on the original $PPI$.
              · If $PPI$ is non-null, set the recognized plan instances ($RPI$) to the result of
                processing the remaining constraints on each element in $PPI$.
              · Add each element of $RPI$ to $PT$ and add each plan it implies to the set of
                potentially implied plans ($PIP$).
      – For each plan $P_j$ in $L$:
          ∗ For any corresponding $PIP_k$ in $PIP$:
              · Set the implied plan instances ($IPI$) to the result of processing implication
                constraints on $PIP_k$.
              · Add $IPI$ to $PT$.

Process-Constraints($CS$ (Constraint Set), $PPI$ (Partial plan instances))

  • For each constraint $C_i$ in $CS$:
      – For each $PPI_i$ in $PPI$,
          ∗ Form the set of new partial plan instances ($NPPI$) that result from binding the
            components in $PPI_i$ that are necessary to evaluate $C$ against elements of $PT$.
          ∗ Form the set of remaining partial plan instances ($RPPI$) that result from evaluat-
            ing $C_i$ on each item in $NPPI$.
      – Set $PPI$ to the concatenation of all the $RPPI$s.
```

Figure 1.2: Our algorithm for automatically recognizing plan instances in code.

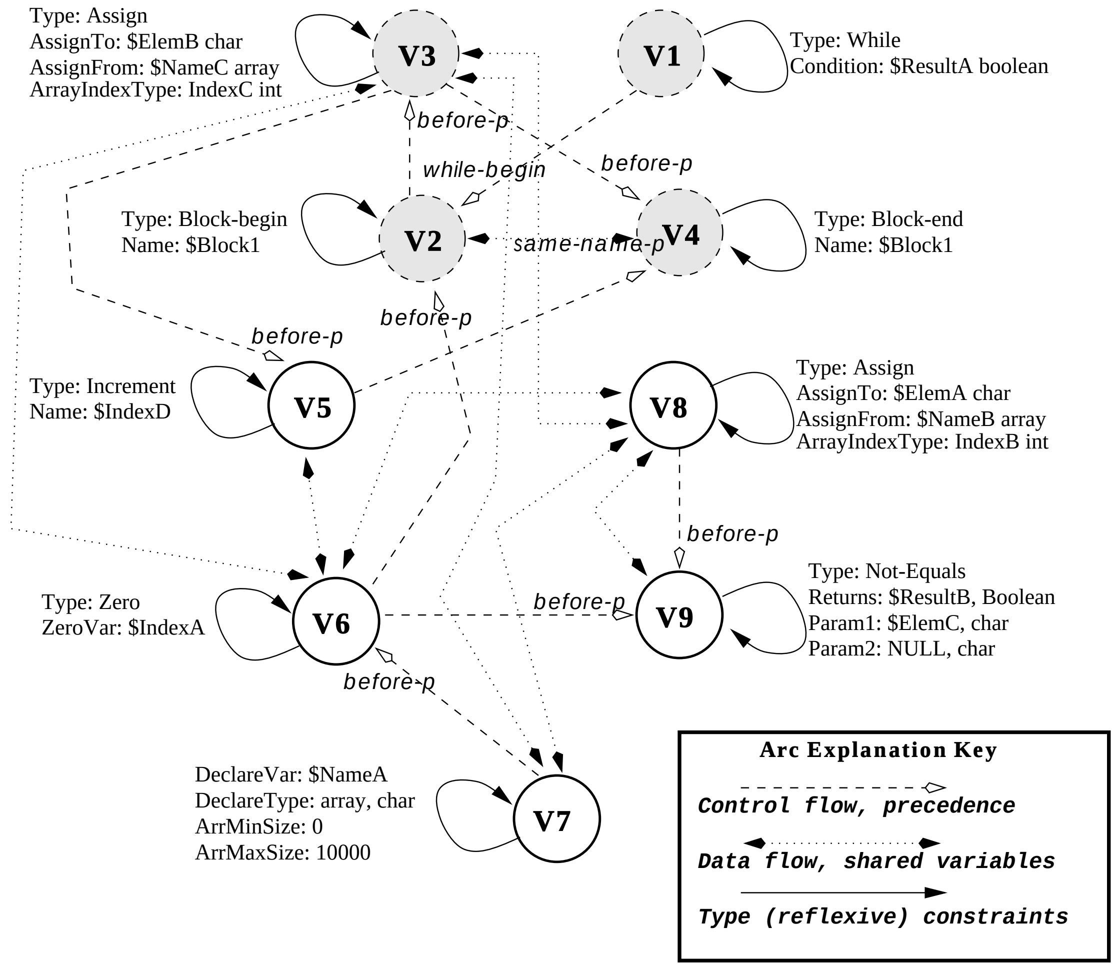


Figure 1.3: An example plan in the MAP-CSP representation.

```
(v3 Assign   (NameC (array (char)))
             (IndexC (int)) (ElemB (char)))
(v1 While    (ResultA (boolean)))
(v2 Begin    (Block1 (block)))
(v4 End      (Block2 (block)))

(before-p    (v1 v3))
(while-begin (v1 v2))
(same-name-p (v2 v4) (Block1 Block2))
(before-p    (v3 v4))
```

Figure 1.4: MAP-CSP representation of code patterns.

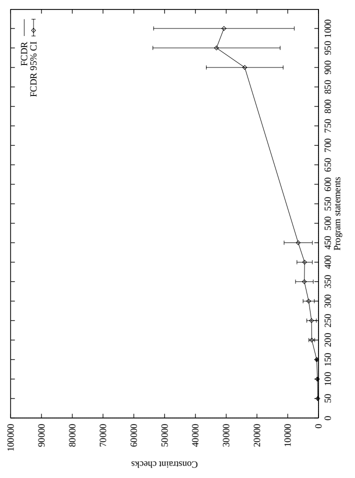


Figure 1.5: Forward Checking, DR (95% conf. interval).

Chapter 2

Program Understanding and Plan Recognition : A Comparative Study

2.1 Introduction

Plan Recognition(PR) is the task of creating a contextual model of the intentions underlying the actions of agents. *Program Understanding*(PU) is the task of creating a contextual model of the intentions underlying actions encoded into program source code. From these simple descriptions of PU and PR, it may be tempting to view PU as simply an instance of PR, and further, recognize that methodologies presented for PR should readily apply to PU. In this paper we clarify the classes of problems that PR and PU methodologies intend to address, and describe the ways in which these classes both differ and resemble one another. As part of this explanation, we show that a straightforward interpretation of PU as a particular kind of PR is incapable of exploiting the particular temporal and causal structures embedded in source code. We point out that PU may be thought of as a *simplified* or more tightly-constrained version of PR that remains NP-hard(Woods & Yang 1996). It additionally provides an interesting example problem on which to build methodologies which may be extended to effectively address the more general PR problem.

This paper is structured as follows. In Sections 2.2 and 2.3 we examine each of PR and PU in turn, and attempt to clarify the structural differences in these problems through use of examples. In Section 2.4 we describe an attempt to model an approach to PU in the spirit of typical PR algorithms, and illustrate the inadequacy of this approach. In Section 2.5 we summarize the main points of this work.

2.2 The Plan Recognition Paradigm

Plan Recognition can be thought of as the task of determining the *best*[1] unified context which causally explains a set of perceived events as they are observed. These events are contextualized within a specific body of knowledge which describes and limits the types and combinations of events that may be expected to occur. This knowledge body is frequently represented as a specialization and decomposition structure of events and actions.

Kautz and Allen(Kautz & Allen 1986) formalized an approach to PR that has served as a primary building block for many subsequent PR methodologies, including (Carberry 1988, Carberry 1990, van Beek, Cohen & Schmidt 1993). PR is defined as the process by which "a set of observed or described actions is explained by constructing a plan that contains them". A model of PR is formed with the intent of both representing actual events or occurences, and of proposing hypothetical explanations of actions. Explaining action through PR involves uncertainty, and therefore it is necessary to somehow recognize some *particular* plan that another agent is performing from a possibly large set of explanatory plans. The process of arbitrating this uncertain selection process is the primary focus of the work of Kautz and Allen, and of plan recognition systems in general.

The general approach of Kautz and Allen is based upon ordinary deductive inference. The *rules* for deduction are rooted in the exhaustive body of knowledge about actions in a particular domain encoded in the form of an **action hierarchy**, as shown in Figure 2.1. The hierarchy depicts specialization relations as dark arrows from specific to general actions. The thinner lines encode decomposition of actions into a set of sub-actions. Not encoded in this figure are additional domain constraints such as temporal relations between sub-actions, although this information is assumed to be available for the plan recognition process. For instance, in **MakePastaDish** it is assumed that the constraint that **Boil** precede **MakeNoodles** is included.

The action hierarchy describes all ways in which any expected action may be performed or used as a step in a more complex action. The *trigger* for deduction is the perception of an action. As an example, observe that the hierarchy encodes that **Boil** and **MakeNoodles** are subactions of **MakePastaDish**, and further that they are subactions of no other action. Perception of an instance of **Boil** then results in the deduction that the more abstract task being undertaken is **MakePastaDish**, and similarly, **PrepareMeal** and **TopLevelAct**. Actions are perceived one at a time, with a model of the agent's intention maintained incrementally following each perception. Although at any point in the process the determination of the perceived agent's plan may be ambiguous[2] (or rather, disjunctive), specific predictions about future activities can still be made. For instance, imagine the perception of an action which is identified as either **MakeSpaghetti** or **MakeFettucini**. Since both of these actions

[1]*Best* is a highly subjective term which changes definition depending on the intent of the particular plan recognition application.

[2]Other work in PR(van Beek et al. 1993) addresses the issue of resolving ambiguity only *when necessary* through interactive dialogue focusing on explanatory plans that share particular faults.

are instances of **MakeNoodles**, we can deduce that the higher level task **MakeNoodles** is being undertaken. Now, since **MakePastaDish** has the additional sub-action **Boil**, we can expect to perceive **Boil** in the future (if it indeed has not yet perceived).

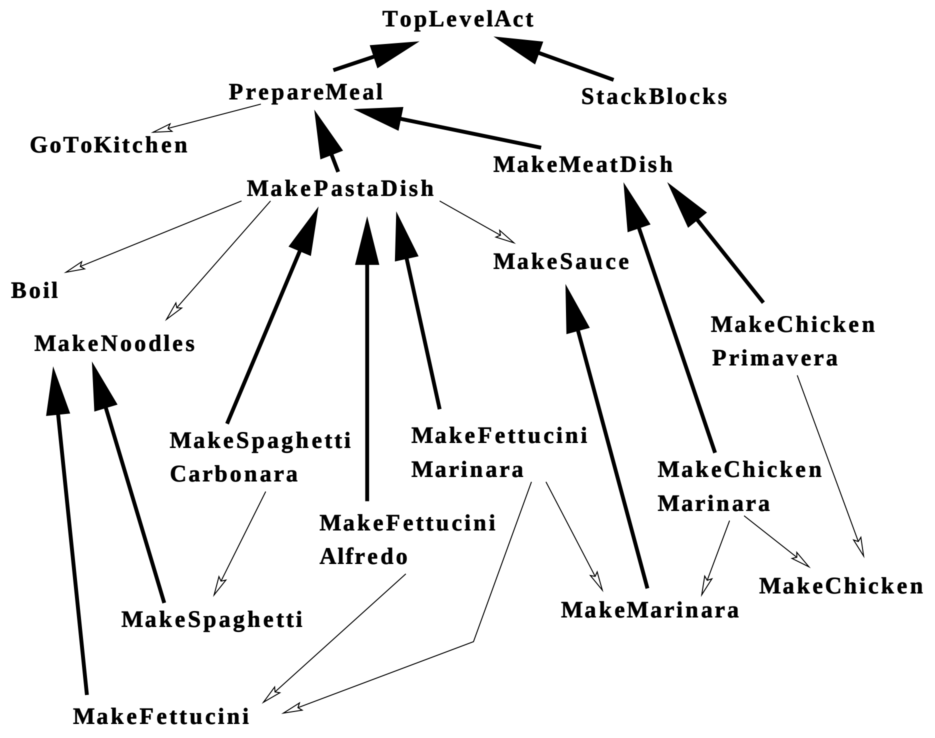


Figure 2.1: Action hierarchy for the cooking domain.

PR as described by Kautz and Allen, embeds several important assumptions:

- *Open Perception*; It must always be assumed that a given set of perceptions of the observed agent or situation may be incomplete. In particular, the perceiver may at any time realize an observation of an act that will result in the need to update current beliefs about the agent's plans.
- *Closed Specialization*; The known ways of performing an action are the only ways of performing that action.
- *Closed Generalization*; All the possible reasons for performing an action are known.
- *Closed Decomposition*; The given decompositions of actions into subactions are the only decompositions.
- *Full Sensibility*; All actions are purposeful, that is, any non top-level action occurs only as part of the decomposition of some top-level action.

- *Simplicity Heuristic or Minimum Cardinality Assumption*; When several actions are observed, assume that the observations are all part of the same top-level act. In general, prefer that as few top level actions occur as possible.

Kautz and Allen explain the plan recognition process as follows. First, the plan hierarchy is processed into a set of axioms according to the hierarchy structure and the assumptions stated. Next, a specialized forward chaining reasoning process embodying a particular inference strategy over these axioms is undertaken. As each observation is received, the system chains up both the abstraction and decomposition hierarchies until a top-level action is reached. The intermediate steps may include many disjunctive statements, such as in the **MakeFettucini**, **MakeSpaghetti** example introduced in Section 2.2. The action hierarchy is used as a *control graph* which directs and limits this disjunctive reasoning. After more than one observation arrives, the system will have derived two or more high-level action instances. The simplicity heuristic is then applied. This heuristic requires that some subsets of these high-level instances be equal, and unifies the disjoint perception interpretations. Exclusive-or reasoning now propagates down the hierarchy, deriving a more restrictive set of assertions about the top-level actions and their subactions. If an inconsistency is detected then the number of top-level acts is incremented and the system backtracks to the point at which the simplicity heuristic was applied.

Kautz (Kautz 1987) identifies two primary problems that must be dealt with in incremental recognition systems. The first of these is the *combinatorial problem* which arises when the minimum cardinality assumption is relaxed to include two or more primary actions. This relaxation allows the number of possible ways of grouping together the set of observations to grow exponentially[3]. The second problem identified is the *persistence problem*: once two observations are tied together or interpreted in a particular context (say as a result of the minimum cardinality assumption), entirely discarding this context simply on account of the arrival of a contradictory piece of information *seems unnatural* from a human reasoning viewpoint.

The simplicity heuristic is the basis for Kautz's model. By minimizing the number of events which account for all observations and accepting this event covering set as the current adopted agent plan, we are describing precisely how to recognize a plan from observation. Consider the following example of the use of minimum event cover in unifiying the contextualization of two action perceptions.

We refer once again to Figure 2.1. A **Boil** action is perceived. **Boil** only occurs as part of **MakePastaDish**, and consequently **MakePastaDish** is adopted as the covering plan. Next, a **MakeMarinara** action is perceived. **MakeMarinara** can be covered by **MakeFettuciniMarinara**, and subsequently **MakePastaDish**, or by **MakeChickenMarinara**, and

[3]Kautz explicitly recognizes that in some domains the combinatorial problem may be largely mediated through various constraints on event types; however, he imagines that in realistically sized problems additional principles will be required. We shall see in our later discussion of PU as a special type of PR, that both action type and other structural problem feature constraints are used in exactly this manner.

subsequently the additional high-level covering plan **MakeMeatDish**. What then is the plan being undertaken? According to minimum cover, **MakePastaDish** covers both **Boil** and **MakeMarinara**, and so **MakePastaDish** is accepted as the current plan; **MakeChickenMarinara** and **MakeMeatDish** may be denied. Now suppose the next perception is a **MakeChicken** action. **MakeChicken** can only be part of a **MakeChickenMarinara** or **MakeChickenPrimavera** action and subsequently the high-level plan **MakeMeatDish** must be inferred. The unified conclusion is forced to include two high level actions now: **MakeChicken** can only be covered by **MakeMeatDish**, **Boil** can only be covered by **MakePastaDish**, and **MakeMarinara** can be covered by either or both of **MakePastaDish** and **MakeMeatDish**. Even this minimization of the high-level actions leaves a great deal of disjunctiveness. For example, it could be the case that **MakePastaDish** covers **MakeMarinara** and **Boil**, and a different chicken dish is being made; or, it could be the case that **MakeMeatDish** covers both **MakeMarinara** and **MakeChicken**, and that a different pasta dish is being made. In fact, it is possible that the **MakeMarinara** action is being *shared* by **MakePastaDish** and **MakeMeatDish**. When such action sharing occurs between plans, additional constraints such as temporal relationships can be very useful in limiting disjunctive conclusions.

Kautz's algorithm

There are three versions of the Kautz approach(Kautz 1987). Each version is based on a different interpretation about how to integrate or group multiple observation explanations and implement the concept of the minimal explanation. The first, *Non-dichronic*, returns the same result independent of the order of observation of events, and identifies the current conclusion as the disjunction of all hypotheses of minimum size. A hypothesis is of minimum size if it involves a minimum number of top-level acts. The second version, *Incremental minimization*, tries to keep the number of top-level acts under consideration to a minimum and only increases when no other option exists. The third version, *Sticky*, prefers to explain each observation by integrating it with the most recently added top-level action.

Kautz's algorithm(Kautz 1987), has three main parts: (1) **ExplainObservation** in which the plan hierarchy is traversed bottom-up, from the observation instance to a top-level plan, giving an independent explanation or explanation set according to all possible disjunctions in the hierarchy; (2) **MatchGraphs** which attempts to merge two independent observation explanation graphs into a single covering explanation graph based on making the "End" or top-level plans involved - a failure to merge signals the need to consider higher-cardinality explanations; and (3) **Group**, which continuously inputs observations and groups them into sets to be explained with independent explanation graphs. From **Group**, a particular minimization function is called, selecting the particular set of explanation graphs which cover all observations.

Recognition summary

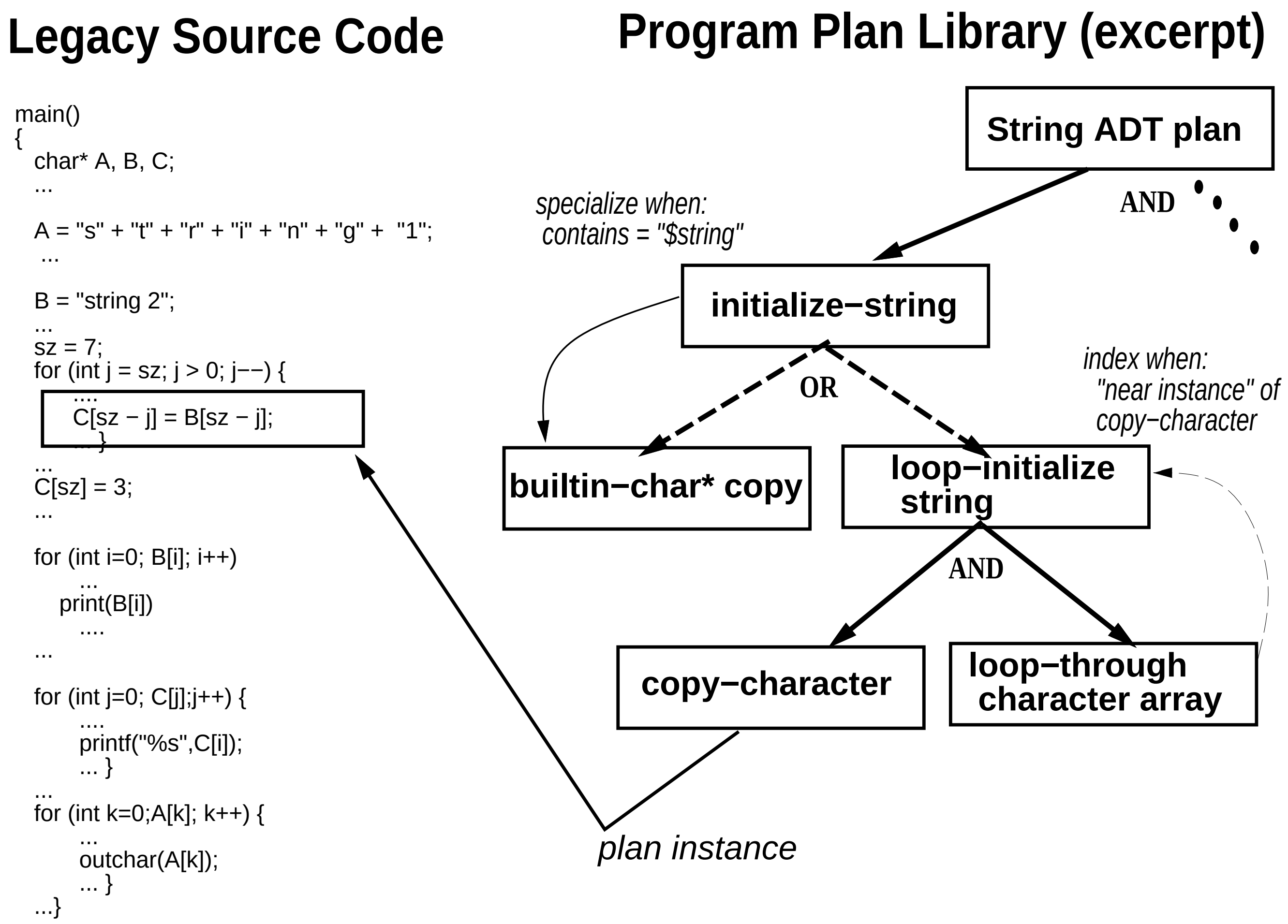
Kautz and Allen's model is designed for iterative or incremental refinement of a model of an agent's plans as successive observations are made of the agent, and as action occurrences are revealed. Following each perception, a possibly disjunctive, non-monotonic model is maintained which hypothesizes the agent's goals. The implicit assumption in this model of plan recognition is that at any time, all observations are not available for deduction. Consequently, non-monotonic conclusions are reached through controlled deduction on the basis of the current, possibly incomplete, observation set.

2.3 Program Understanding Overview

Program understanding is the attempt to construct a (possibly partial) *mapping* between the expert's store of relevant knowledge structures and components inherent in the source code. This mapping may be viewed as the task of determining the *best* unified context which causally explains a well-structured set of known program source statements—essentially, trying to infer which programming plans were instantiated by the actions in the program.

Researchers in PU(Woods & Yang 1995*b*, Woods & Yang 1995*a*, Woods & Yang 1995*c*, Quilici 1994, Quilici & Chin 1994, Quilici 1995, Quilici & Chin 1995, Kozaczynski & Ning 1994, Rich & Waters 1990, Wills 1990, Wills 1992, Rugaber, Stirewalt & Wills 1995) have tended to take approaches to PU based on the existence of a domain-dependent knowledge library which consists of programming plan templates and concepts. Source code is interpreted within the context of a specific body of knowledge that describes how programs in general, and domain-programs in particular, are known to be structured. Various top-down and bottom-up search strategies are utilized to construct partial mappings between the legacy source and knowledge. Notable examples include Quilici(Quilici 1994), Kozaczynski and Ning(Kozaczynski & Ning 1994), Rich and Waters(Rich & Waters 1990) and Wills(Wills 1990, Wills 1992). To some extent, these approaches are all aimed at improving the effectiveness of the mapping process through exploiting heuristic knowledge. Recent work in PU(Woods & Yang 1995*c*, Woods & Quilici 1995) has been motivated by a desire to bring together the range of program understanding strategies and heuristics into a single representational framework.

In Figure 2.2, a subset of expert knowledge about a particular application domain is represented in a fragment of a hierarchical library of program templates. The encoded structure, or *knowledge constraints*, include temporal, control flow, and data flow relations among the components of plans. In addition, typical or expected structure can be represented in the hierarchy as preferences for certain common specializations or indices for frequently related plans. Figure 2.2 shows one possible mapping between a plan template from the library and a specific legacy source fragment, in this case a single source statement. The existence of such a mapping essentially *explains* the presence of the low-level source statement at a higher level of abstraction, in this case as an instance of the plan template **copy-character** specified in the library.



```
main()
{
  char* A, B, C;
  ...
  A = "s" + "t" + "r" + "i" + "n" + "g" +  "1";
   ...
  B = "string 2";
  ...
  sz = 7;
  for (int j = sz; j > 0; j--) {
      ....
      C[sz - j] = B[sz - j];
      ... }
  ...
  C[sz] = 3;
  ...
  for (int i=0; B[i]; i++)
      ...
     print(B[i])
      ....
  ...
  for (int j=0; C[j];j++) {
       ....
       printf("%s",C[i]);
       ... }
  ...
  for (int k=0;A[k]; k++) {
       ...
       outchar(A[k]);
       ... }
  ...}
```

Figure 2.2: Conceptualizing source with a plan library.

As in PR, PU frameworks make several key assumptions about the domains in which they work and the task in general, including:

- *Closed Perception*; The source program under consideration at any point in time, together with any derived structural constraints, makes up all of the perceptual information that will be available. In particular, it will never be the case that a program statement or part that was absent in the previously encountered functional specification will be perceived. Although the focus of PU may be only a sub-part of a larger program, the part in question is itself complete.
- *Closed Specialization*; The known ways of specializing a particular abstract plan are the only ways to consider, despite the fact that others may exist.

- *Open Generalization*; All possible reasons for performing a particular source statement or abstract plan can never be known, however the known specializations are the only ones of interest in constructing partial explanations.

- *Open Decomposition*; The given decompositions of plans into subplans are only a subset of all possible decompositions in any domain, however, the known decompositions are the only ones of interest in constructing partial explanations.

- *Partial Sensibility*; All source statement actions are purposeful, that is, any recogznied non-top-level plan occurs only as part of some top-level plan. However, this top-level plan may not reside in the knowledge hierarchy. Further, program statements will necessarily exist that cannot be explained with the partial knowledge hierarchy.

Just as plan recognition has adopted simplicity measures as a way of dealing with combinatorial problems in explaining the relationship between two or more observed actions, PU work has attempted to adopt preferences based on various types of locality. In particular:

- *Ordinary spatial locality*; Programs exhibit spatial and temporal locality. That is, statements that are spatially *tend* to be related to one another with a higher likelihood than those that are spatially distant.

- *Temporal locality*; If one were to observe a program's execution trace, it would be possible to recognize patterns of commonly executed program parts. These patterns could be used to identify possibly related program parts based on previously collected knowledge about the way in which various program parts inter-relate.

- *Functional locality*; Programs can be statically decomposed into abstract syntax trees annotated with control and data flow information. This additional structure greatly strengthens the notion of spatial locality, in that relatedness is made explicit rather than inferred. In contrast to the *weak* or preferential constraints indicated by other localities, this structure can be thought of as *strong* constraints. The ability to check correspondences between such structure and expected relations embedded in the hierarchical knowledge library, provides an excellent source of information to use in reducing the combinatorics of explanation.

In (Woods & Yang 1995*c*, Woods & Yang 1995*b*, Woods & Yang 1995*a*, Woods & Quilici 1995, Woods & Yang 1996) we present results as part of a ongoing project, the purpose of which is to demonstrate that an effective approach to partial PU is possible with large legacy code examples. Additionally through this project we wish to concisely and precisely represent the particular problem structures, constraints, and solution strategies in a unified framework. The model we have chosen for this generalized representation of PU is as a *Constraint Satisfaction Problem (CSP)*(Mackworth 1977).

A Constraint Satisfaction Problem[4] typically consists of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints among the variables which restrict domain value assignments. A solution of a CSP is a set of domain value-to-variable assignments such that all inter-variable constraints are satisfied. Much research has been done in creating algorithms for solving CSPs, ranging from global(Kondrak & van Beek 1995) and local search-based methods(Sosic & Gu 1990, Minton et al. 1992, Yang & Fong 1992), constraint-propagation problem simplifications(Nadel 1989, Dechter 1992, Prosser 1993), and hybrid combinations of these approaches. The general approach to representing PU as constraint satisfaction involves two CSPs:

- The **program understanding CSP**, or PU-CSP in which a large legacy source is first divided into spatially and functionally related blocks, and each block is explained in terms of the existing program plan hierarchy. It is important to note that the entire problem may be considered hierarchically: program blocks may be relatively *large*, as is the case with a program procedure, or *small*, as is the case for program statements. A solution to a PU-CSP "explains" source blocks *globally* in terms of mappings to a program library such that the knowledge (library) and structural (source code) constraints are satisfied.

- The abstract program plan **template matching CSP** or MAP-CSP. We view MAP-CSP as an integral part of the more ambitious understanding task. Successful matches "locally explain" certain program blocks, and these local solutions can then be exploited to restrict the larger PU-CSP. A MAP-CSP or program template matching problem can be stated as follows: given a plan template with a number of elements and constraints among the elements, find all instances of the template in a source code. As an example, consider finding all instances of a particular abstract data type in a piece of legacy source. A template is defined in terms of a set of variables corresponding to features of the data type. Structural properties of the data type are represented as inter-feature constraints for temporality, data-flow, control-flow and element typing. Each variable has a *domain* ranging over all possible source program statements. A solution to the MAP-CSP "explains" a *local* subset of program statements in the source through identification of a mapping between the template features and source statements. Each assignment must satisfy all knowledge and structural constraints.

Understanding summary

The PU models we have described are designed for non-iterative or all-at-once *partial* explanation of a source code given a particular program plan library. The primary source of information is program source annotated for control flow, data flow and other statically available information that strongly constrains explanations for any source element. The

[4]See (Kumar 1992) for an accessible and detailed treatment of Constraint Satisfaction Problems.

implicit assumption in this model is that a great deal of contextual information is available in advance of any plan recognition. Consequently, explanatory conclusions are reached through controlled reduction of sub-part explanations, by exploiting the fact that structural constraints must match knowledge constraints for consistent explanations.

In other work, we intend to demonstrate the feasibility of PU-CSP in partially understanding very large legacy code. Towards this goal, we have successfully implemented and tested MAP-CSP for several representative examples(Woods & Yang 1995*a*, Woods & Yang 1995*c*). This empirical work is a step towards demonstrating that the MAP-CSP representation and algorithm is capable of providing all-instance results in moderately-sized program blocks. An efficient MAP-CSP algorithm will significantly reduce the difficulty of the larger PU-CSP algorithm. We observe that for legacy source examples of up to approximately 500 lines of code, even relatively straightforward strategies located all instances of the ADT in approximately one minute of CPU time. In examples of up to 300 lines of code, all instances were identifiable in approximately 20 seconds.

2.4 Program Understanding as Plan Recognition

It is apparent that the PR and PU problems are closely related. In particular, a solution to either problem must be based upon mapping sets of actions to elements of hierarchical libraries of plans in a consistent fashion. As we are concerned with identifying good solution strategies for program understanding, and since a large body of work has been produced based on the generic plan recognition strategy of Kautz and Allen, a natural question to ask is whether or not this algorithm can be applied directly to PU.

Consider the following simple legacy source code fragment:

$$\ldots\ \{\ \ldots\ c := a + b;\quad print(c);\quad c := c/2;\ \ldots\quad \}\ \ldots$$

We interpret this example as the following simple series of observed actions: **Sum**, **Print**, and **Divide**. We ignore assignment and other structural constraints for this example. We wish to contextualize the example source with respect to the hierarchy of Figure 2.3. Consider that each program statement constitutes a block or program action that must be explained in the context of the hierarchy.

Consider what Kautz's approach would suggest as the unified plan after encountering each of the three observations. After the first observation of **Sum**, the disjunctive result would be **Sum-P-and-Q** *is-part-of* **Sum-Values** *is-part-of* **(Print-Sum-Plan or (Find-Avg-Plan** *is-part-of* **Print-Avg-Plan))**. The second observation of **Print** would result in an independent explanation of (1) **Print-Value** *is-a* **Print-Sum** *is-part-of* **Print-Sum-Plan**, or of (2) **Print-Value** *is-a* **Print-Temp-Result** *is-part-of* **Sum-Values** *is-part-of* **Find-Avg** *is-part-of* **Print-Avg-Plan**, or of (3) **Print-Value** *is-a* **Print-Average** *is-part-of* **Find-Avg** *is-part-of* **Print-Avg-Plan**. A *minimal* covering graph for this observation set would be rooted in **Print-Avg-Plan** which covers both observations with cardinality 1. The third

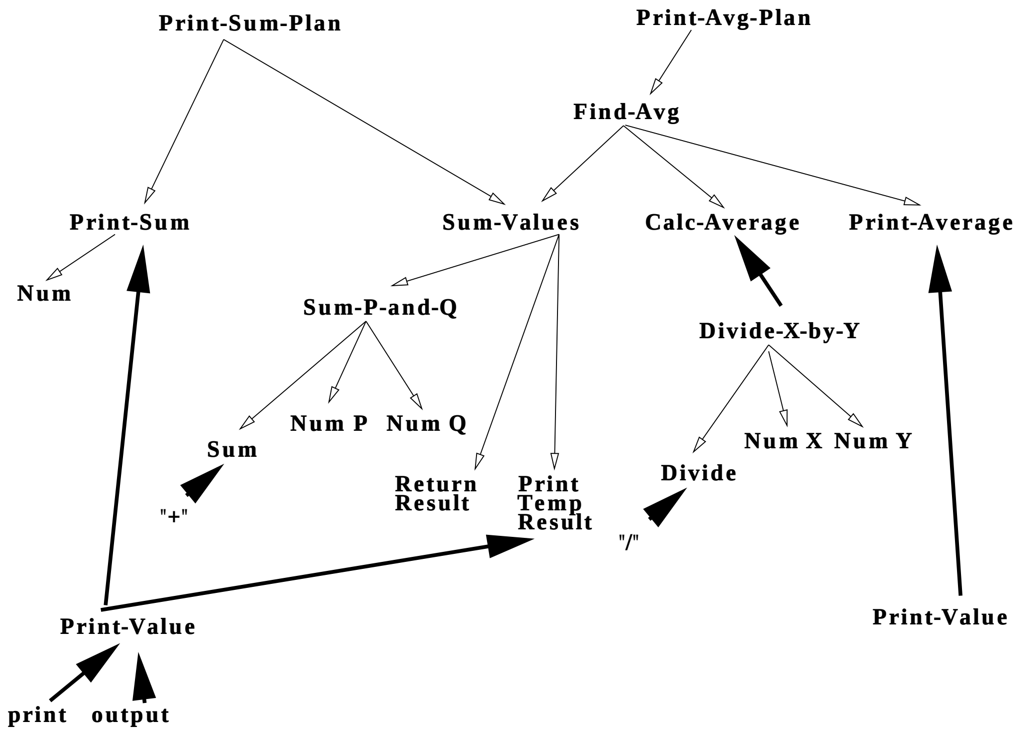


Figure 2.3: Example program plan hierarchy.

observation is **Divide-by-2**. This would result in an independent explanation of **Divide-X-by-Y** *is-a* **Calc-Average** *is-part-of* **Find-Avg** *is-part-of* **Print-Avg-Plan**. The minimal covering graph for this observation set would be rooted in **Print-Avg-Plan** once again, since this plan covers the observation set with cardinality 1. This covering explanation "explains" the **Sum** instance as an instance of **Sum-P-and-Q** in **Sum-Values**, **Print** as an instance of **Print-Temp-Result** in **Sum-Values**, and **Divide** as **Calc-Average** in **Find-Avg**.

This example demonstrates the inappropriateness of allowing a covering set to cover more actions than have been encountered. Although the explanation is minimal in terms of top-level actions, it allows for the assumption that future actions will be encountered. This difficulty is obvious in the above example where the **Print** corresponds to the **Print-Average** sub-plan. With advance knowledge that there are in fact *no more applicable observations* (as is the case with PU), this assumption is not justified. Consequently, a non-minimal covering would be the correct (possibly partial) explanation. In addition, any partial recognition algorithm must account for the fact that it may not be possible to root every low-level perceived action to a top-level plan, much less the same top-level plan or minimal set of plans.

We discuss the similarity and difference of PR and PU approaches according in terms of the following points:

- The **plan library** is assumed to be complete or incomplete at the time of reasoning

Observation Set $\downarrow$: Library Structure $\rightarrow$	**InComplete**	**Complete**
InComplete	Extended PR Work	Kautz PR Algorithm
Complete	Partial PU	Special case PU

Table 2.1: PU versus PR Comparison of assumptions.

or during the deductive step.

- The **observation set** is assumed to be complete or incomplete at the time of reasoning or deduction.

- The set of observations or perceived actions is either **strongly** or **weakly** causally constrained. For example, in PU there is no ambiguity regarding the causal relatedness of some actions (such as in the case of actions in a loop structure), while in PR for the cooking domain, action perceptions require additional assumptions to connect them causally. This difference is of great importance in controlling the combinatorial problem Kautz outlines: the *structural constraints* available through preprocessing of legacy source are what support efficient solution of large PU problems. It is important to note that Kautz makes use of this type of source constraint where possible, in the form of observable temporal relationships in the cooking domain.

In particular, we categorize PU and PR approaches based on their assumptions about hierarchy and observation set completeness. These results are highlighted in Table 2.1, and discussed as follows:

- **Library InComplete, Observation Set InComplete**

  The Kautz approach is inapplicable to this most general case due to the assumption that a complete library is integral to Kautz's minimal cover approach. Some more recent PR work(Spencer 1991, Cohen & Spencer 1993) has attempted to extend PR in this direction. Program understanding approaches capable of admitting partial understanding (such as the constraint-based approach) are applicable in any library-incomplete situation. Program understanding typically fails to address the case of incomplete observations *except* in the case of partial understanding situations, where it is explicitly understood that a code **fragment** is the source input. In these partial PU cases, it may be assumed that the incomplete observation sets are locally (spatially and functionally) connected and consequently exhibit the same degree of structural constrainedness as a complete observation set. In the cases of partial recognition where it is expected that the source will map to only some subset of the library, no difference in behaviour will be expected for a partial, functionally complete source.

- **Library InComplete, Observation Set Complete**

  Once again Kautz's approach requires a complete library assumption, and in this case we have an additional difficulty: Kautz's PR algorithm assumes that the observation set is incomplete and that recognition is done incrementally. We have seen that the PR minimal cover can be incorrect if one applies an incremental algorithm to a complete observation set. Partial program understanding algorithms apply in this situation.

- **Library Complete, Observation Set InComplete**

  At any point in time of any reasoning about plan-based explanation of behaviour, this situation is the precise expectation of PR approaches such as that of Kautz. Non-monotonic decisions or interpretations are made after each successive observation, anticipating that another observation will be forthcoming. Most program understanding work is based upon a strong assumption that it is impossible to completely specify a sufficient program plan library to cover all program source, even in a limited domain. Some approaches (such as the constraint-based and memory-based(Quilici 1994) approaches) can make strong claims about their ability to recognize the correct plans in cases where a complete library is known in advance.

- **Library Complete, Observation Set Complete**

  It would appear that this most strongly constrained situation would admit the most constrained algorithms as a result. However, Kautz's approach will not apply here. For instance, a minimal set covering can imply the existence of observations that are in fact not present at any point during the incremental approach. This situation is in fact a special case for PU. One may view this as the case where an attempt is being made to recognize source code generated solely through automated use of the library. Thus, the library completely covers the source by definition.

2.5 Conclusion

We have seen that PR and PU approaches exhibit a great deal of similarity:

- PR and PU strategies share a representation of *understanding* as the successful construction of a mapping between hierarchical pre-existing knowledge libraries and some input observation set.
- Both strategies attempt to reduce the combinatorial difficulties of integrating multi-observation explanation by exploiting available *knowledge* constraints on action composition as required temporal ordering of sub-actions.

However, the approaches differ in very significant ways:

- The Kautz PR strategy assumes a complete library and incomplete observation set, and consequently is inapplicable to a more restricted PU domain in which an incomplete library and complete observation set are the norm.
- The differing assumption sets can result in over-committed solutions when the PR concept of obsevation set minimal covering is applied to PU. The assumption of an incomplete observation set is the basis for preferring few top-level plans rather than a number of apparently disjoint partial plans.
- PR has a less-restrictive constraint set upon which to limit the combinatorial problem of disjunctive explanation. While PU may exploit the wealth of structural constraints easily-extracted from the source before recognition, PR is limited to explicit temporal constraints. Consequently, we expect to solve larger PU problems more efficiently than comparably sized PR problems.
- PU can be thought of as a special, well constrained, case of PR which remains difficult (NP-hard). While we have seen why general PR approaches are inapplicable to typical PU problem instances, it should be emphasized that one important result of this study is the suggestion that the techniques used in PU be considered for the more general PR problem. In particular, certain PR problem instances could admit pre-processing of the observation set to identify particular causal relationships. These explicit relationships should be applied in conjunction with action representations so as to increase the number and type of constraints available in the problem solution.

Acknowledgments

Alex Quilici would like to acknowledge partial support for this work by the KBSA project, Air Force Rome Labs, under Air Force contract #F30602-93-C-0257. Steven Woods and Qiang Yang would like to acknowledge the Natural Sciences and Engineering Research Council of Canada and the Institute for Computer Research (ICR) for their support. Thank you to Stephanie Ellis for useful comments on earlier drafts of this work.

Bibliography

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.

Cohen, R. & Spencer, B. (1993), Specifying and updating plan libraries for plan recognition tasks, Technical Report cs-93-10, University of Waterloo.

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

Freuder, E. & Wallace, J. (1992), 'Partial constraint satisfaction', *Artificial Intelligence* **58**, 21–70.

Hartman, J. (1991), Understanding natural programs using proper decomposition, *in* 'Proceedings of the International Conference on Software Engineering', Austin TX, pp. 62–73.

Johnson, W. L. (1986), *Intention Based Diagnosis of Novice Programming Errors*, Morgan Kaufman, Los Altos, CA.

Kautz, H. (1987), A Formal Theory of Plan Recognition, PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York.

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

Kondrak, G. & van Beek, P. (1995), A theoretical evaluation of selected backtracking algorithms, *in* 'Proceedings of the 14th International Joint Conference on Artificial Intelligence', pp. 541–547.

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

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

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

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

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.

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. (1993), A hybrid approach to recognizing programming plans, *in* 'Proceedings of the Working Conference on Reverse Engineering', Baltimore, MD, pp. 126–133.

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

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

Quilici, A. & Chin, D. (1994), A cooperative program understanding environment., *in* 'Proceedings of the Ninth Knowledge-Based Software Engineering Conference', Monterey, CA, pp. 125–132.

Quilici, A. & Chin, D. (1995), DECODE: A cooperative environment for reverse-engineering legacy software, *in* 'Proceedings of the Second Working Conference on Reverse-Engineering', IEEE Computer Society Press, pp. 156–165.

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

Rugaber, S., Stirewalt, K. & Wills, L. (1995), The interleaving problem in program understanding, *in* 'Proceedings of the Second Working Conference on Reverse-Engineering', IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos CA 90720-1264, pp. 166–175.

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

Spencer, B. (1991), Assimilation in Plan Recognition via Truth Maintenance with Reduced Redundancy, PhD thesis, University of Waterloo.

van Beek, P., Cohen, R. & Schmidt, K. (1993), 'From plan critiquing to clarification dialogue for cooperative response generation', *Computational Intelligence.*

von Mayhrhauser, A. & Vans, A. M. (1995), 'Program comprehension during software maintenance and evolution', *IEEE Computer* pp. 44–55.

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

Wills, L. M. (1992), Automated program recognition by Graph Parsing, PhD thesis, MIT.

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

Woods, S. & Quilici, A. (1995), Representing memory-based program understanding as constraint satisfaction, Technical report, University of Waterloo, Department of Computer Science.

Woods, S. & Yang, Q. (1995*a*), 'Constraint-based plan recognition in legacy code', *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE).*

Woods, S. & Yang, Q. (1995*b*), Program understanding as constraint satisfaction, *in* 'Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE)', pp. 318–327.

Woods, S. & Yang, Q. (1995*c*), Program understanding as constraint satisfaction: Representation and reasoning techniques, Technical report, University of Waterloo, Department of Computer Science.

Woods, S. & Yang, Q. (1996), Approaching the program understanding problem: Analysis and a heuristic solution, *in* 'Proceedings of the 18th International Conference on Software Engineering', Berlin, Germany. To appear.

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