Automated Software Engineering, 5, 347–372 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Applying Plan Recognition Algorithms To Program Understanding[*]

ALEX QUILICI alex@wiliki.eng.hawaii.edu
Department of Electrical Engineering, University of Hawaii at Manoa
2540 Dole St, Holmes 483, Honolulu, HI 96822

QIANG YANG qyang@cs.sfu.ca
School of Computing Science, Simon Fraser University
Burnaby, British Columbia V5A 1S6

STEVEN WOODS sgwoods@spectra.eng.hawaii.edu
Department of Electrical Engineering, University of Hawaii at Manoa
2540 Dole St, Holmes 483, Honolulu, HI 96822

Abstract. Program understanding is often viewed as the task of extracting plans and design goals from program source. As such, it is natural to try to apply standard AI plan recognition techniques to the program understanding problem. Yet program understanding researchers have quietly, but consistently, avoided the use of these plan recognition algorithms. This paper shows that treating program understanding as plan recognition is too simplistic and that traditional AI search algorithms for plan recognition are not suitable, as is, for program understanding. In particular, we show (1) that the program understanding task differs significantly from the typical general plan recognition task along several key dimensions, (2) that the program understanding task has particular properties that make it particularly amenable to constraint satisfaction techniques, and (3) that augmenting AI plan recognition algorithms with these techniques can lead to effective solutions for the program understanding problem.

Keywords: program understanding, constraint satisfaction, performance evaluation

1. Introduction

Program understanding is often described as the process of recognizing program plans in source code (Woods and Yang, 1995; Quilici, 1994; Kozaczynski and Ning, 1994; Wills, 1990; Johnson, 1986). In particular, most program understanding algorithms explicitly use a library of programming plans, along with various heuristic strategies, to locate instances of these plans in the code. Because the program understanding task is so closely related to plan recognition, one would expect to see researchers directly apply well known plan recognition algorithms to the task (Kautz, 1987; Kautz and Allen, 1986). However, they have not, and have instead chosen to develop their own special purpose algorithms. This paper is an attempt to understand and explain why.

Program understanding can be divided into two general categories: *precise* and *imprecise*. *Precise* program understanding occurs when a program understanding mechanism recognizes in a program every instance of a particular plan.[1] That is, there are no false

[*] This work was partially supported by the KBSA project, Air Force Rome Labs, under Air Force contract #F30602-93-C-0257, NSF, under research grant #IRI-9309795, the Natural Sciences and Engineering Research Council of Canada, and the Information Technology Research Centre.

positives (incorrectly recognized instances of a plan) and there are no false negatives (incorrectly missed instances of a plan). This level of understanding is what is required for correctness preserving automatic program transformations. *Imprecise* program understanding allows the program understanding mechanism to guess that a particular plan is present or to miss instances of a plan defined in the library. This type of understanding is useful as a starting point for further human exploration of the code, but is not suitable for automatic transformation.

In this paper, we examine the relationship between plan recognition and *precise* program understanding and study the assumptions underlying each task. As part of this analysis, we present an approach to program understanding in the spirit of typical plan recognition algorithms, and we illustrate the inadequacy of this approach. We then demonstrate how a constraint satisfaction-based approach to plan recognition is particularly well-suited to this type of program understanding, and we show how one existing AI plan recognition algorithm can be modified to take this into account. Finally, we discuss the practical relevance of plan-based program understanding to real-world software engineers working to reverse engineer legacy systems.

Our motivation for this work is to help move program understanding from being an isolated subproblem of AI into the mainstream of AI research. This will allow results in AI involving plan recognition and constraint satisfaction to be quickly integrated into our program understanding algorithms, and it will allow work in program understanding to influence the general AI community and perhaps benefit other AI application areas.

2. An AI Approach To Plan Recognition

Plan recognition is the task of determining the *best*[2] unified *context* which causally explains a set of perceived events as they are observed. A context is essentially a hierarchical set of plans and goals that accounts for the observed actions. This process generally assumes a specific body of knowledge which describes and limits the types and combinations of events that may be expected to occur.

Kautz and Allen (Kautz, 1987; Kautz and Allen, 1986) formalized an approach to plan recognition that has served as a primary building block for many subsequent plan recognition methodologies, including (van Beek et al., 1984; Song and Cohen, 1991). They provide a general algorithm by which “a set of observed or described actions is explained by constructing a plan that contains them”. In particular, as actions are observed, hypothetical explanations are proposed for them. This process involves uncertainty, as at any time there are a number of candidate explanations for an action, but only a portion of the actions within each of those candidates may have been observed. 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.

Kautz and Allen’s approach 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*. This action hierarchy describes all ways an action may be performed or used as a step in a more complex action.

2.1. An Example Action Hierarchy

Figure 1 is an example action hierarchy.

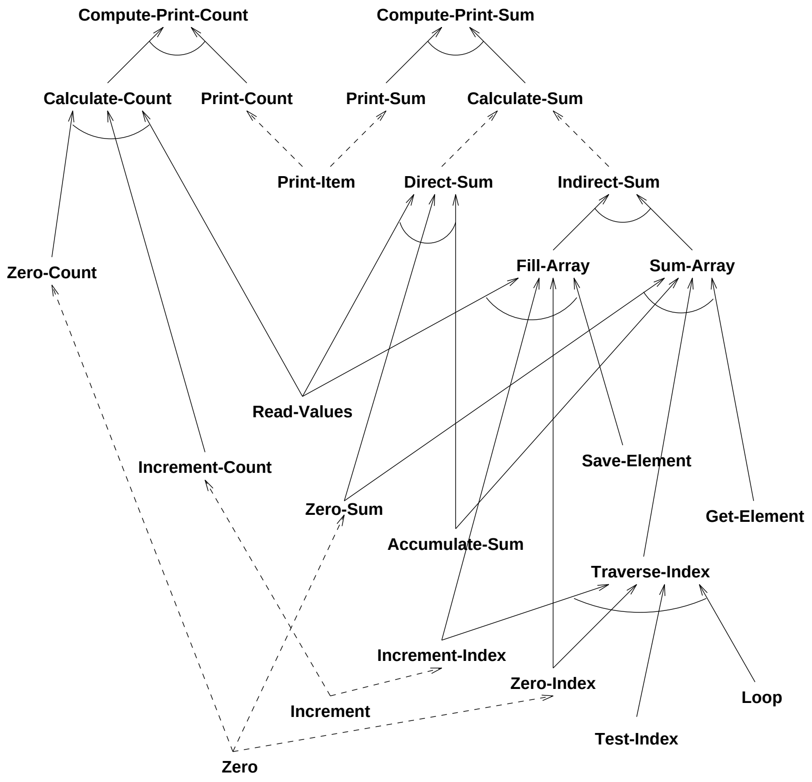

Figure 1. An example action hierarchy.

This hierarchy is sufficient to understand the three code fragments in Figure 2. Figure 2(a) sums the input values as each is read, Figure 2(b) first reads them into an array and then computes the sum by traversing the array, and Figure 2(c) sums and counts the input values as each is read.

An action hierarchy captures specialization and decomposition relationships between actions. In Figure 1, the specialization relationships are illustrated using a dashed line and the decomposition relationships using a regular line.

Specialization relationships between actions capture the notions that there are multiple ways to perform a given task and that a given action can be used to perform multiple tasks. For example, **Direct-Sum** and **Indirect-Sum** are two alternative ways to accomplish **Calculate-Sum**, and **Print-Item** is one way to accomplish both the **Print-Sum** and the **Print-Count** actions. *Decomposition relationships* represent the situation where a plan requires a set of actions. For example, **Compute-Print-Sum** requires both a **Print-Sum** and a **Calculate-Sum**.

```
sum = 0;
while (scanf("%i", &value) == 1)
  sum += value;
printf("%i\n", sum);
```

(a) *Directly compute and print the sum of the input values.*

```
n = 0;
while (scanf("%i", &value) == 1)
  a[n++] = value;
sum = 0
for (i = 0; i < n; i++)
  sum += a[i];
printf("%i\n", sum);
```

(b) *Use an array to compute and print the sum of the input values.*

```
n = 0;
sum = 0;
while (scanf("%i", &value) == 1)
{
  sum += value;
  n++;
}
printf("%i %i\n", sum, n);
```

(c) *Directly count and sum input values.*

Figure 2. Some examples of C code our hierarchy can be used to understand.

In addition, although not shown, the action hierarchy also captures constraints between these actions. For example, in **Compute-Print-Sum**, there's a constraint that the sum computed by **Calculate-Sum** must be the one displayed by **Print-Sum**.

2.2. Using The Action Hierarchy

The Kautz and Allen approach starts by turning the action hierarchy into a set of axioms that captures the structure of the hierarchy and its underlying assumptions. The actual recognition process then undertakes a specialized forward chaining reasoning process over these axioms.[3]

Figure 3 contains this algorithm. As the program understander encounters each action, it chains up the action hierarchy until it reaches a top-level plan. The result is a set of possible paths from the observed action to top-level actions. These paths constitute an initial set of possible explanations (in terms of higher-level plans) of the action.

After more than one observation arrives, the system will have derived two or more sets of paths to top-level action instances (that is, it will have found a set of paths from each observed action, through the action hierarchy, to top-level actions). It then forms the cross-product of these explanation graphs, which constitutes an initial set of disjoint explanations for all actions so far, and it forms additional possible explanations by attempting to merge

```
Algorithm: KautzRecognize($Obs_{set}$, $Hier$);

Input: A set of observations $Obs_{event} \in Obs_{set}$, a plan hierarchy $Hier$ structure, with $Obs_{first}$
being the first observation;

Output: A hypothesis $Hypo$ consisting of a set of instantiated portions of $Hier$ that cover the set
of observations $Obs_{set}$;

SubRoutines
A.  $ExplainObservation(Obs_{event}, Hier)$: returns a set of explanations $ObsEventGraph$
      which explains $Obs_{event}$.
B.  $AddHypo(Hypo, NewExpl)$: returns a new hypothesis by adding a hypothesized
      explanation for an event, $NewExpl$, to an existing hypothesized explanation for the
      previous events, $Hypo$, without merging.
C.  $MergeHypo(Hypo, NewExpl)$: returns the set of hypotheses generated by merging a
      hypothesized explanation for an event, $NewExpl$, with a hypothesized explanation for
      the previous events, $Hypo$.

Main Routine
1   $Hypo_{set} := ExplainObservation(Obs_{first}, Hier)$;
2   $Obs_{set} = Obs_{set} - Obs_{first}$;
3   for each $Obs_{event}$ in $Obs_{set}$ do
4     $NewExpl_{set} := ExplainObservation(Obs_{event}, Hier)$;
5     $NewHypo_{set} = NULL$;
6     for each $Hypo$ in $Hypo_{set}$ do
7       for each $NewExpl$ in $NewExpl_{set}$ do
8         $NewHypo_{set} :=$
            $NewHypo_{set} + AddHypo(Hypo, NewExpl) + MergeHypo(Hypo, NewExpl)$;
9       endfor (step 7)
10    endfor (step 6)
11    $Hypo_{set} = NewHypo_{set}$;
12  endfor (step 2)
13  return $Hypo \in Hypo_{set}$ with minimum $cardinality(Hypo)$;
```

Figure 3. The Kautz non-dichronic plan recognition algorithm.

these disjoint explanations into a single explanation. The merge is done by trying to locate and bind a shared top-level ancestor and by using constraints to eliminate inconsistent merged explanations. Finally, it uses a “simplicity heuristic” to determine the best merged explanation. This heuristic is to prefer as few high-level actions as possible or, in other words, to prefer the explanation for a set of actions that has a minimal set of higher-level plans that “cover” them.

2.3. An Example Using The Action Hierarchy

We illustrate the Kautz and Allen algorithm by showing how it uses the hierarchy in Figure 1 to understand the program in Figure 2(a).

The first action is a **Zero**. Figure 4 shows the five possible explanation graphs for this action. All of these explanation graphs lead to a single top-level action, which varies depending on the particular explanation graph, so the system's simplicity heuristic can't determine a single best explanation.

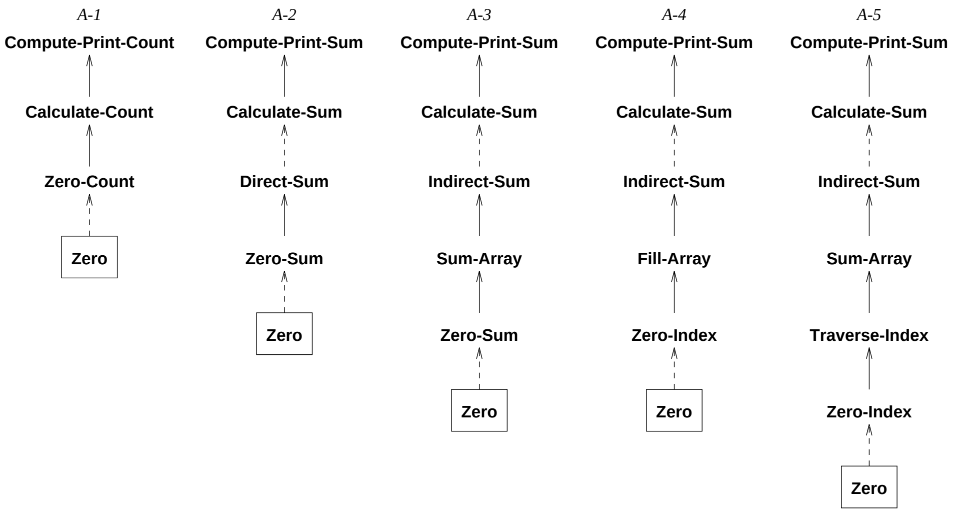

Figure 4. The possible explanations for a **Zero** action.

The second action is a **Read-Values**. Figure 5 shows the three possible explanation graphs for that action.

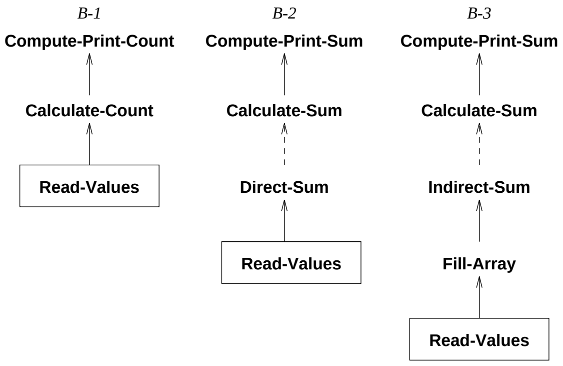

Figure 5. The possible explanations for a **ReadValues** action.

The system must then try to form the cross product of these explanation graphs, merge them, and determine the preferred explanations. The result of this process is a set of explanation graphs, which have either **Compute-Print-Count**, **Compute-Print-Sum**, or both as their set of top-level nodes. Figure 6 shows several of the merged explanation graphs that result.

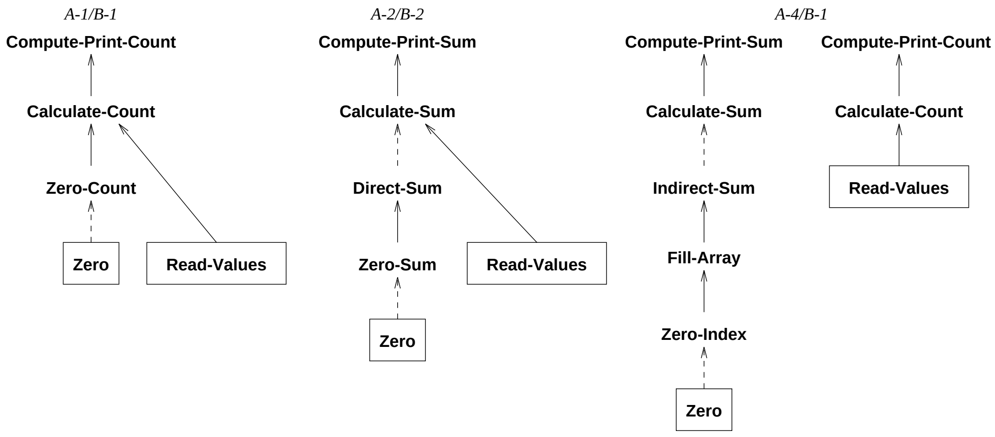

Figure 6. A few of the possible merged explanations for the first two actions.

The simplicity heuristic says to prefer the explanation graphs with the fewest top-level nodes, which in this case are those that have a single top-level node, either **Compute-Print-Count** or **Compute-Print-Sum**. The heuristic does not, however, specify which of these top-level nodes is the most appropriate explanation.

The third action is an **Accumulate-Sum**. After the system generates the two explanation graphs that contain it and merges them with the explanation graphs for the previous actions, the result is once again a set of explanation graphs. Again, the simplest explanation graphs are those with a single top-level node, and these explanations all happen to have **Compute-Print-Sum** as their single top-level node. As a result, **Compute-Print-Sum** is understood as the high-level plan. Figure 7 shows several of these “simple” explanations.

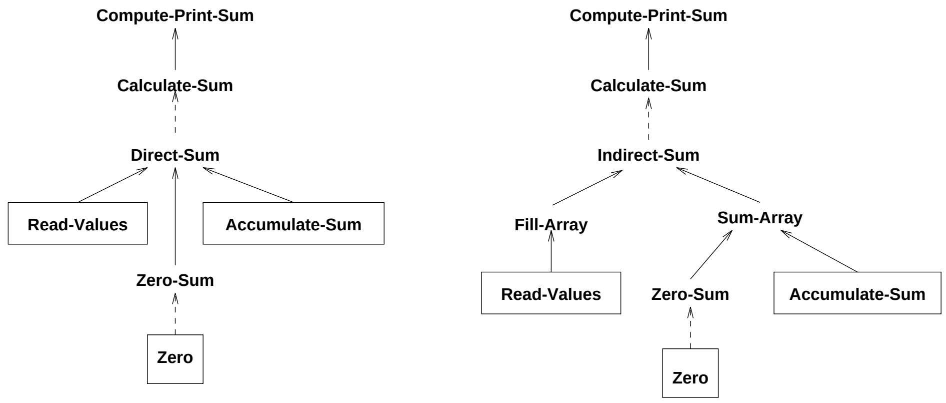

Figure 7. A few of the possible merged explanations for the first three actions.

The final action is a **Print-Item**. There are two possible explanation graphs for it, and merging these with the set constructed to understand the previous actions leads to a final set of explanation graphs. Once again, there are explanation graphs terminating in a single top-level node, and these explanation graphs all share **Compute-Print-Sum** as that node. Figure 8 shows several of these final "simple" explanations, one of which we would consider the most reasonable explanation graph.

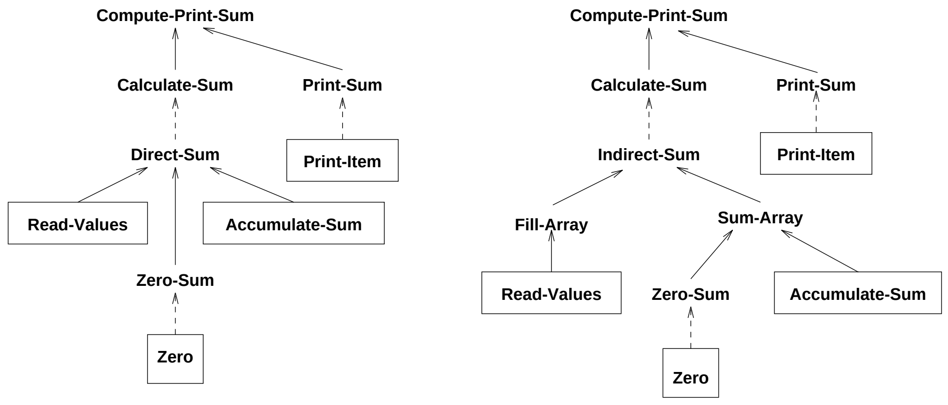

Figure 8. A few of the possible merged explanations for all four actions.

This "simplicity" heuristic is key: by minimizing the number of hypotheses which account for all observations and accepting this event covering set as the current plan, we describe precisely how to recognize a top-level plan from low-level observations.

3. Problems With Applying AI Plan Recognition To Program Understanding

The Kautz and Allen approach to plan recognition is elegant and the basis for much subsequent work in plan recognition. Given that program understanding appears to be a form of plan recognition, it's worth considering whether this approach is applicable to program understanding.

One key difference between plan recognition and program understanding is that plan recognition assumes *Open Perception* and program understanding assumes *Closed Perception*. That is, at any point in time, the plan recognition algorithm has an incomplete set of observed actions and, as a result, the plan recognizer is making a best guess as to what plan is present, and most of its work is in coming up with this best guess. In contrast, in program understanding exactly the opposite is true. The source program under consideration, together with any derived structural constraints, makes up all of the perceptual information that will ever be available. That is, it will never be the case that a program action that was absent in the previously encountered functional specification will appear later. Although the focus of program understanding may be only a sub-part of a larger program, the part in question is itself complete.

3.1. Incorrect Plan Recognition

As a consequence of its open perception assumption and the simplicity heuristic used to deal with it, the Kautz and Allen approach can find an incorrect explanation, despite there being sufficient knowledge to eliminate it as a candidate.

To illustrate, consider removing the `printf` statement (the **Print-Items** action) from Figure 2(a), leaving only the **Zero**, **Read-Values**, and **Accumulate-Sum** actions. As we just saw, after seeing the **Accumulate-Sum**, the Kautz and Allen algorithm will conclude that **Compute-Print-Sum** is the appropriate top-level explanation. However, if the **Print-Item** that this plan requires never occurs, the problem is that this explanation cannot be the case. The explanation, in fact, should be limited to **Calculate-Sum**, and should not include the top-level node at all.

It is also possible for the Kautz and Allen algorithm to select a very misleading explanation graph based on the idea of minimal cover. Figure 9 contains the most appropriate explanation for the fragment in Figure 2(c) (which counts and sums its input values) and Figure 10 contains the minimal cover explanation graph determined by Kautz and Allen's algorithm.

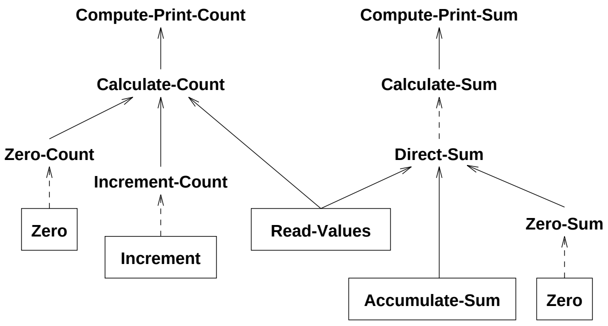

Figure 9. The most appropriate explanation for the input counting and summing fragment.

Unfortunately, this minimal cover explanation is that the user is filling and summing an array, and it relies on the possibility that actions such as accessing and assigning array elements will be encountered later. The problem is that this explanation is wrong, given that we know no more actions relevant to these plans will appear in the program. Although this explanation is minimal in terms of top-level actions, it allows for the assumption that future actions will be encountered. Here, in fact, the knowledge that certain actions will not occur dramatically changes the understanding of the code.

The idea of "minimal cover" is intended to be an application of Occam's principle: prefer the explanation that requires the fewest assumptions. In particular, it minimizes the number of top-level actions the recognizer is assuming the user is trying to accomplish. However, each individual explanation is assuming that certain actions that haven't yet appeared will eventually appear, and these assumptions aren't being taken into account. This problem suggests that other interpretations of Occam's principle could readily be applied to plan

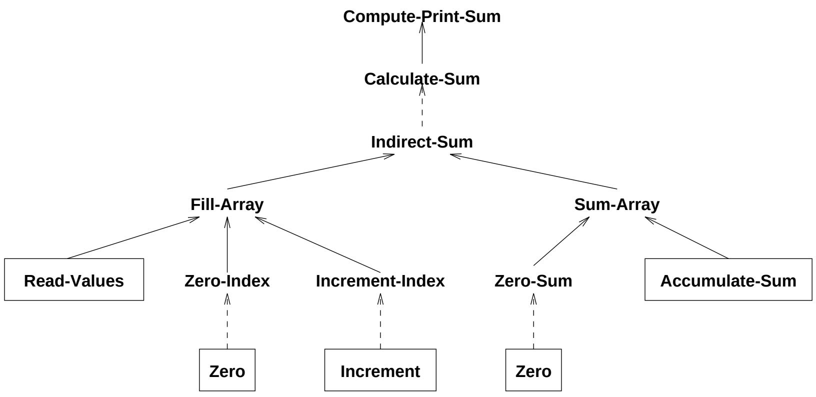

Figure 10. The incorrect explanation minimal cover produces for the input counting and summing fragment.

recognition. One alternative, for example, is to prefer the top-level plans with the fewest missing actions.

In fact, in program understanding it is inappropriate for the covering set to ever cover actions than have not been encountered. Consequently, an exact covering set that is not necessarily minimal would give the correct explanation. To make the distinction more precise, let E be a set of observed events; in program understanding, E is the set of program statements. Let $\mathcal{H}$ be the set of hypotheses where the search is performed. These are the program plans at various levels of detail. The goal of both plan recognition and program understanding is simply to find one or more subsets of hypotheses H from $\mathcal{H}$ such that H “covers” E.

The two problems differ in how H is defined. We can understand a hypothesis as the set of events that it covers. Then Kautz’s principle can be understood as finding a smallest set of hypotheses H_K such that

$$H_K \supseteq E$$

In contrast, program understanding can be defined as finding a smallest set of hypotheses H_P such that

$$H_P = E$$

The difference is that in the latter, the cover must be exact. The hypotheses must explain all observed events, but no more. We call this the *principle of minimal exact-coverage*

Situations like the ones in these examples can occur frequently in program understanding because of *incomplete plan libraries*. It is unlikely that a plan library will contain all the plans necessary to understand a program (Chin and Quilici, 1996; Quilici, 1995). **Calculate-Sum** and **Calculate-Count**, for example, might also be part of a **Calculate-Average** plan, which may not actually occur in the plan library. The result is that any algorithm we use must be capable of producing a forest of intermediate plans and should not attempt to infer potentially incorrect high-level groupings.

It is reasonable to wonder whether Kautz's approach can easily be modified to address this problem. Figure 11 shows the architecture of one such modification.

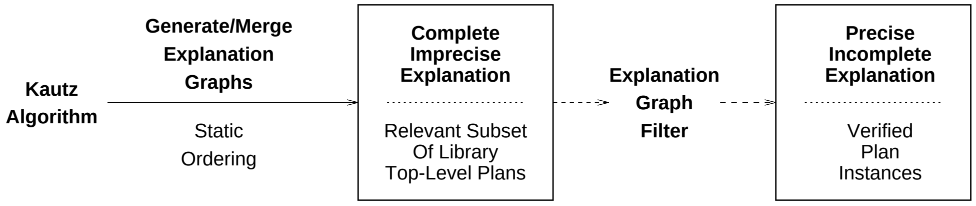
Kautz Algorithm
Generate/Merge Explanation Graphs
Static Ordering
Complete Imprecise Explanation
Relevant Subset Of Library Top-Level Plans
Explanation Graph Filter
Precise Incomplete Explanation
Verified Plan Instances

Figure 11. The architecture of an extended version of Kautz's algorithm.

This modification is to add an explanation filter that, after all events have been processed, runs through *all* of the hypothesized explanations and eliminates any nodes that cover an event that did not occur. This guarantees that the algorithm never terminates with an incorrect explanation. However, *all* hypothesized explanations must be examined, not just those with the minimum "cover". In the explanation shown in Figure 10, for example, pruning any node that covers an absent action leaves no recognized plans. However, there are other explanations, such as that shown in Figure 9, that do contain plans that cover only actions that are present. These plans will be recognized after pruning these explanations.

The Kautz and Allen algorithm carries along every candidate hypothesized explanation, so this is a relatively straightforward change to make. It's computational cost, however, depends on how many hypothesized explanations must be computed and carried along during the understanding process. The next section discusses this issue.

3.2. Inefficient Plan Recognition

Another consequence of the open perception assumption of the Kautz and Allen approach is that it implies that events must be processed in the order they are encountered, which is not necessarily the most efficient event ordering. The approach provides no mechanism to take into account alternative orderings that might reduce the number of hypothesized explanations generated.

In fact, the Kautz and Allen algorithm may compute and maintain an exponential number of hypothesized explanations. Suppose the program's abstract syntax tree has N actions: A_1, A_2, ..., A_N. Given the first two actions, A_1 and A_2, and their respective sets of stand-alone explanations, the Kautz algorithm does two things: it generates the cross product of these explanations, and it attempts to form *additional* merged explanations.[4] If, on average, an action A_i has K possible explanations, the cross product alone will generate up to K^2 new explanations. Thus, the cost of generating these explanations, C_1, will be at least K^2 (this ignores any explanations generated by the merge). The algorithm then repeats the process with these explanations and the explanations for the next action, A_3. The cost of this merge, C_2, is at least $K \times K^2$ or K^3 explanations. After repeating this process for subsequent syntax tree entries, the number of explanations is at least $\sum_{i=1}^{n-1} C_i$, which is $K^2 + K^3 + \ldots + K^N$, or $O(K^N)$, explanations. As a result, the Kautz algorithm is in some cases at least exponential in N, which is the number of program actions. This problem is especially relevant to program understanding since most programs involve many thousands of actions (or more).

How large is K? It's clearly dependent on the structure of the plan library. To estimate the size K, we assume a plan library that is organized in d rows, with the first row being the top-level explanations and a bottom row that corresponds to program actions. If we assume a uniform connection, so that each entry in the first row connects to two nodes in the second row (as a component in two plans or as a specialization of two plans), every node except the top-level node has two parents. In this case, each leaf node has 2^d paths leading from the node to a root node, so $K = 2^d$. As a result, in that case the actual worst-case complexity of the Kautz Algorithm is no better than $O(2^{d^N})$.

Determining an overall explanation for a set of events can be viewed as proceeding through a search space of explanation graphs. That is, the algorithm must repeatedly generate and merge explanation graphs. In plan recognition, there are several key factors that contribute to the size of the search space and the effectiveness of the search. One is the overall complexity of the plan hierarchy. Specifically, K (the average number of different explanations for a given observation) is affected by the number of nodes in the plan hierarchy and the number of plans in which a typical action or plan is a component (the out-degree of nodes in the graph). Another factor is the effectiveness of constraints. Specifically, C_i (the effort in generating and merging explanations), is affected by how many potential merges of explanation graphs fail based on constraints, lessening the number of potential explanations that need to be formed. A final factor is the relatedness of the events being understood. The less related an event is to previously understood events (in terms of its explanations sharing common ancestors with previous explanations), the more explanation graphs will need to be generated.

To lower the cost of understanding a set of events, we must minimize the overall number of explanation graphs formed. Unfortunately, the complexity of a plan hierarchy and the overall effectiveness of its constraints is fixed. That leaves only one option for reducing the size of the search space: ordering the events so that the events that are most closely related and have the most effective constraints are processed first. The problem is that the Kautz algorithm processes events in a single, static ordering that is fixed by the order in which observations appear. For program understanding, this static ordering is the order in which statements appear in the source. However, as a result of the closed perception assumption, program understanders have all events available and are therefore free to process these events in any order. This observation suggests exploring ways to most effectively order the processing of events so as to constrain the size of the search.

4. Modifying Plan Recognition Approaches To Support Program Understanding

One way to characterize plan recognition approaches that are derivatives of the Kautz and Allen algorithm is to say that they process actions in order, try to hypothesize *complete* explanation chains that cover each action, and use subsequent actions to shrink the set of explanations (when the actions can be combined under some high-level action) or hypothesize additional explanations (when they can't). At the end of a pass through all actions, the plan recognizer has a set of preferred *hypothesized* explanations for those actions.

In some sense, program understanding has more knowledge available than is assumed by the Kautz and Allen approach to plan recognition. In particular, program understanders have the complete set of actions that are present in the program. This allows program understanders to process actions in any order, including orders determined by the structure of the plan library, not simply the order in which they appear in the program. They can hypothesize and verify *partial* explanation chains that directly cover an action, rather than forming and later discarding complete explanation chains. And they can form a precise understanding by repeating this process, gradually constructing explanations chains from verified partial explanations.

In addition, the program understanding domain provides detailed data-flow and control flow constraints between its actions. The prevalence of these inter-action constraints should reduce the size of the overall search space that must be processed, as compared to the wide open domains with which the Kautz and Allen approach was designed to deal.

4.1. The Hierarchical MAP-CSP Approach

We have developed an approach to program understanding that takes full advantage of the closed perception assumption. The resulting algorithm is shown in Figure 12.

Algorithm: HIERARCHICAL, LIBRARY-DRIVEN VERIFICATION(Obs_{set},$Hier$)

Input: A set of observations, $Obs_{event} \in Obs_{set}$ and a set of plans $P_1, P_2, \ldots, P_n$, where each P_i is composed of a set of components, $C_{i_1}, C_{i_2}, \ldots$, and the plans are organized into a set of layers, $L_1, L_2, \ldots, L_k$, such that layer L_1 contains only those plans whose components are AST entries, and where any other L_i contains only components that are in layer L_1 through L_{i-1}.

Output: A set of instantiated plans from the hierachy that are verified to be present in the program.

SubRoutines

A. $Verify(Plan)$: return instances of plans that are present in the program

B. $PotentialInstances(P_i)$: return TRUE if each C_{i_j} of P_i has at least 1 instance in Obs_{set} or $Recognized_{set}$.

Main Routine

```
1   $Recognized_{set} := NULL$;
2   for each $L_i$ in $L_1 \ldots L_k$ do
3     for each $P_j in L_i$ do
4       if $PotentialInstances(P_j)$ then
5         $NewInstances_{set} := Verify(P_j)$
6         if $NewInstances_{set} != NULL$ then
7           $Recognized_{set} := Union(Recognized_{set}, NewInstances_{set})$;
8     endfor (step 3)
9   endfor (step 2)
10  return $Recognized_{set}$;
```

Figure 12. Hierarchical Bottom-Up Program Understanding.

There are two key ideas in this algorithm. The first insight is that given a hypothesized plan that contains only actions (and not sub-plans), we can immediately verify whether that plan actually exists by locating the plan's actions and verifying its constraints. For example, suppose we have a counting plan consisting of an assignment, a comparison, and an increment, where the comparison is a loop test and the increment is in the loop body. Since all its actions are AST (abtract syntax tree) entries, we can immediately search the program for occurrences of this plan. As a result, we can use each action in the AST as an index to the set of potential plans that might contain it and then check whether each of these plans are present. At the end of a pass through all actions, the plan recognizer has located all verified single-plan explanations for each action.

The other insight is that one way to locate complete, verified explanation chains is to organize the plan library in layers, where the first layer is those plans that consist solely of events in a program's AST, the next layer is those plans that depend only on the events in the AST and plans in the first layer, and so on. An example of a bottom-layer plan is **Double**, the plan of doubling a value, which is implemented either using an AST multiply (multiplying by 2) or add (adding a value to itself). Similarly, any plan combining **Double** and other AST entries is in the next layer. As a result, after recognizing those plans in the initial layer, the plan recognizer can run through each of these plans and verify whether the plans in the next layer that can contain them are actually present, creating a new set of verified recognized plans. This process is repeated until there are no newly recognized plans.

The key question is how can we perform this process of verifying a hypothesized plan. That is, given that an action suggests a set of possible plans that might explain it, how can we verify which of these plans are actually present? Given the presence of many constraints between the actions in any plan, this suggests using a constraint satisfaction approach.

4.2. MAP-CSP: Verifying A Hypothesized Plan

A Constraint Satisfaction Problem[5] (CSP) 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 variable that restrict domain value assignments. A solution for a CSP is a set of domain value to variable assignments such that all inter-variable constraints are satisfied, and there exist a large variety of methods to choose from for solving CSPs (Woods, 1996; Kondrak and van Beek, 1995; Prosser, 1993; Dechter, 1992; Freuder and Wallace, 1992; Minton et al., 1992; Sidebottom and Havens, 1992; Yang and Fong, 1992; Sosic and Gu, 1990; Nadel, 1989; Mackworth et al., 1985).

We can use a CSP for the task of verifying whether a single plan is present in the following way:

- Each plan action is a variable in the CSP representing it. The set of domain values for each variable is the set of source statements and sub-plans that have already been recognized.

- The type of each action corresponds to a reflexive (node) constraint on the variable representing it.

- The constraints between actions (such as required data- and control-flow relationships) correspond to inter-variable (arc) constraints.

Various example plans using this representation can be found in (Zhang, 1997; Quilici and Woods, 1996; Woods and Yang, 1995).

A solution to this CSP consists of the mapping of all assignments of plan actions to source code statements, where each assignment must satisfy all constraints. This solution provides a mapping that *explains* the matched source statements as parts of one or more plan instances. We call the CSP engine for recognizing all instances of a program plan *Map-CSP*, and the repeated application of Map-CSP on a plan library organized in layers, hierarchical *Map-CSP* (*H-Map-CSP*).

This CSP-based approach differs somewhat from traditional AI plan recognition in that the search for explanatory plans is now library driven rather action driven. That is, rather than taking an action and searching for a plan that explains it, we are instead taking a plan and trying to determine which actions it explains.

4.3. *Efficiency Expectations For H-Map-CSP*

The H-Map-CSP algorithm's complexity is $O(N * C_{MAP-CSP})$, where N is the number of plans in the library and $C_{MAP-CSP}$ is the complexity of verifying whether there are instances of a hypothesized plan present. While $C_{MAP-CSP}$ in the worst case is still exponential (Woods and Yang, 1996), in practice, there is reason to believe this algorithm will perform far better than Kautz.

As shown in Figure 13 H-Map-CSP moves directly toward the target of precise understanding, without first forming an imprecise understanding and then trying to refine it. At a macro-level, the algorithm reorders and groups events so that they are considered only in terms of a hypothesized explanation in which they are related. At a micro-level, constraint satisfaction algorithms explicitly relax the temporal ordering of domain ranges (e.g., events) by dynamically re-arranging the domains (in the spirit of some types of forward checking algorithms); in effect, trying to reap the benefits of improved search results through more effective constraint applications which reduce entire sub-parts of the search space.

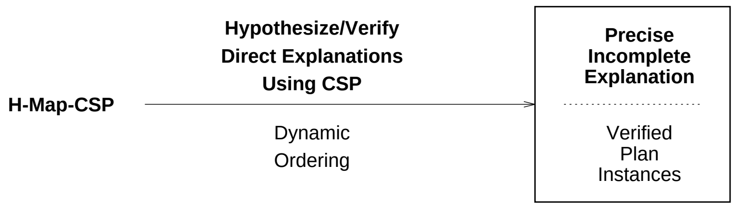

Figure 13. The architecture of our approach to precise program understanding.

5. Some Experimental Evidence

The previous section has shown how we can derive a new approach to program plan recognition by examining an existing AI plan recognition algorithm, studying its assumptions, determining how these assumptions differ from the program understanding problem, and then modifying this approach to take advantage of the differences. While it's clear that our new plan recognition approach to program understanding addresses the correctness issue, and there's clearly potential to address the efficiency issue, it's necessary to carry out experiments to determine whether the efficiency issue is, in fact, addressed.

5.1. An Initial Experiment

As an initial experiment, we generated a set of test programs and applied the constraint satisfaction approach to locate instances of a given plan. That is, we assumed that a plan has been suggested by the presence of a program action and then empirically verified how efficient or inefficient it is to recognize *all* instances of the plans containing this action (or any action of the same type appearing later in the program).

Our mechanism for generating programs was to start with a particular plan instance (or set of instances) and to randomly add program statements surrounding each instance according to a pre-determined distribution of program statement types (Woods, 1996). We used this approach to form a collection of programs of the desired sizes.

Why have we chosen to work with artificially generated programs rather than real-world programs? Our primary motivation has been that we wanted to focus solely on the scalability of the recognition algorithm as programs with similar characteristics grow in size. In particular, we wanted to keep the distribution of AST components, the particular plans we were trying to locate, and the likelihood of finding those plans constant across different program sizes. That's difficult if not impossible to do with real-world programs. While we can certainly find real-world programs of varying sizes, the distribution of their components and the particular plans they contain will vary significantly.

While our approach runs the obvious danger of generating artificial programs that are far divorced from real-world programs, we have tried to mitigate this problem in several ways. First, we are generating programs containing plans that frequently appear in real-world programs, such as traversing arrays or strings. And second, we are generating programs according to a "standard" distribution of statements that corresponds to what we've found in student C programs.

Our test programs ranged in size from of 50 to 6,000 lines in size, with 10 different programs at each size. Based on the results of solving 10 CSP problems at each size level, we generate a 95% confidence interval for the number of constraint checks occurring during the search.

As in earlier experiments with smaller programs, we used *Forward Checking with Dynamic Rearrangement* (FCDR) as our particular method of solving constraint satisfaction problems (Quilici and Woods, 1997). Our measure of efficiency is the number of constraint checks performed, as constraint checking is where the dominant amount of work occurs in an attempt to recognize a program plan.

Figure 14 shows the results of running our initial experiment. The plan instances we tested had an average of approximately 10-15 components and 20-25 constraints.

Essentially, the results show a curve in which our standard distribution increases from 5,000 constraint checks for 1,000-line programs, up to 65,000 constraint checks for 5,000-line programs. While this curve appears to be potentially exponential in nature, there is at least one reason to remain optimistic: the steepness of the curve may be an artifact of our

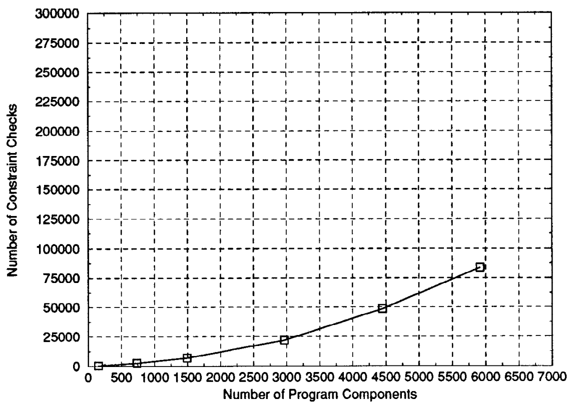

Figure 14. The results of a representative experiment with generated programs.

particular method of representing programs. In particular, our experiments rely primarily on the equivalent of control-flow constraints and do not have data-flow constraints. Since data-flow constraints tend to be much more restrictive than control-flow constraints, they have the potential to reduce the steepness of the curve significantly and extend the size of the programs to which we can apply our plan recognition algorithm. In essence, we can view our initial results as showing the potential for the CSP-based approach with only minimal structural constraints in the programs being understood.

5.2. An Experiment With A "Real-World" Program

We would like to know whether the data-flow constraints present in real-world programs will lead to a performance improvement in our constraint-based approach. One obvious experiment is to search for our existing plan in a collection of real-world programs of various sizes and measure the constraints evaluated in the process. However, this approach has several practical problems. One is that to have meaningful scaling results, we want to vary the size of the programs but not the distribution of components, the plan we are trying to locate, or the data-flow and control-flow complexity. Another is that our current home-built C data-flow and control-flow analyzer can only handle a subset of the language and can't compute data-flow relationships that cross function boundaries.

To address these problems, our initial real-world experiment works with variants of a single function that computes some basic statistics. That is, we have taken the core of the function's body and replicated it varying numbers of times to generate programs of different sizes from 50 lines to 5,000 lines (Zhang, 1997). We then searched for the same conceptual array traversing plan we searched for in our original set of experiments.[6]

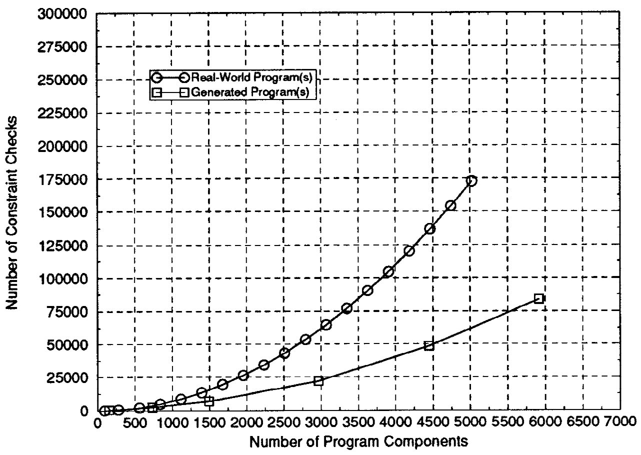

Figure 15. The results of an experiment with "real-world" programs.

Figure 15 shows the results of running this new experiment overlaid on the results shown in Figure 14. Unfortunately, the results are exactly the opposite of what we would have predicted. In applying our plan recognition engine to real-world programs we have a much more sharply rising curve.

There are several potential explanations for this unexpected increase. One is that while we are using similar plans, there is a substantially different component distribution between our artificial and real-world programs. For example, there were roughly twice as many assignments in the real-world program as in the artificial program, and one-fifth as many tests. Our earlier results have shown that the distribution of program components (equally distributed versus some components appear far more often than others) makes a significant difference in the performance of the recognizer (Woods and Quilici, 1996). Another is the possibility that our particular CSP algorithm is not fully exploiting the tightness of real-world constraints. That is, there may be information that has been computed in the data-flow graph of the program that is not being used to reduce the size of the search space.

5.3. *An Experiment With A Two-Phase CSP Algorithm*

One key to solving CSPs quickly is to narrow down the sets of domain values for the nodes participating in any given constraint. That's because the constraint may be evaluated once for each pair of values in the cross product of the domain values of those nodes. To see why, suppose we are evaluating a constraint C that holds between two variables A (with domain values $A_1, A_2, \ldots, A_m$) and B (with domain values and $B_1, B_2, \ldots, B_n$). A CSP essentially evaluates this constraint by checking whether the constraint holds for each A_i,B_j pair. As a result, the more we can reduce the size of the sets of domain values, the more we can reduce the complexity of the search space. A common technique for doing so is to have a constraint propagation phase before attempting to find a solution to the CSP. That is, the idea is to set up the CSP, use constraint propagation to try to shrink the domain values of the CSP, and only then to attempt to find a solution to the CSP.

It appears that we can augment MAP-CSP with a form of domain-specific constraint propagation. That is, we can add a single preprocessing step that exploits information gleaned from static analysis to eliminate some of the domain values. This step requires that we extend the data-flow graph representing the program to group a node's successor and predecessor nodes by their type.[7] For example, suppose we have a particular statement, S, that uses a set of program variables x_1, x_2, . . ., x_n. Its predecessors are the statements that placed values into each x_i (e.g., assignments, additions, and so on) and its successors are the statements that use each x_i. This extension allows to take a particular node N (e.g., a multiplication) and a type T (e.g., addition) and immediately find all of N's predecessor or successor nodes of type T (e.g., the particular additions that produced the values that were multiplied, or the various additions that made use of the multiplication's results).

We can then make use of this information by adding a set of node constraints to our CSP variables. In particular, if we have a data-dependency constraint between two plan actions, A and B, such that B depends on A for some variable V, we can add a node constraint to A that it must have at least one successor of type B, and a node constraint to B that it must have at least one predecessor of type A. These node constraints can be quickly evaluated before processing any inter-variable constraints. In particular, for each domain value of A, A_i, we can now determine in constant time whether there exists any value of type B in its successor set. If not, it means that there is no way A_i can have a data dependency on any instance of B, and we can eliminate A_i from A's domain values. Similarly, for each node B_j, we can in constant time determine whether there exists any value of type A in its predecessor set. If not, we eliminate B_j from B's domain values.

As a result of this insight, we have modified our approach to include this additional filtering step, after setting up the CSP and before attempting to solve it, and we have rerun our previous experiments starting with the CSP that results after this filtering. Figure 16 shows the results of this experiment.[8]

As expected, this filtering reduced the amount of work done by the CSP engine to the point where we could reasonably run experiments on 10,000 line programs. It provides evidence for our earlier hypothesis that the data-flow information in real-world programs is useful in constraining the amount of work done by the plan recognition engine.

6. Future Work

There are several key areas for us to explore in the future, each of which is an entire research project in its own right.

6.1. Additional Experimentation

Our initial results are promising, however, they're based on a small set of experiments with a collection of artificially generated C programs and a small group of carefully constructed C programs. In addition, we have a made a variety of simplifying assumptions in terms

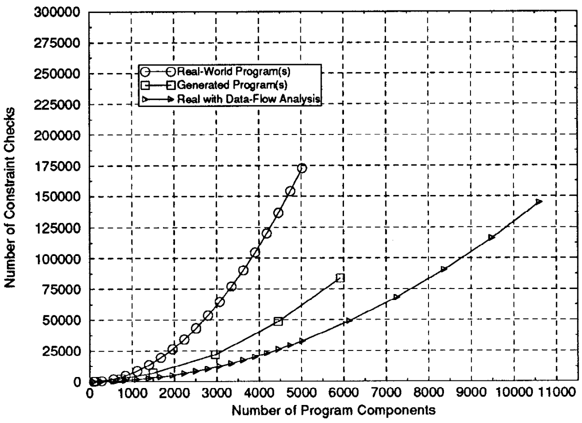

Figure 16. The results of our experiment with domain value pre-filtering.

of the language constructs that appear in the programs we understand, the overall structure of these programs, and the type of control and data flow information that's available to use from these programs. It's an open question whether these assumptions, which have arisen from deficiencies in our home-grown control and data flow analysis tools, change the empirical performance of our algorithm. As a result, it's necessary for us to obtain more powerful analysis tools and use them to apply Map-CSP to explore the performance of searching for individual plans in real world C code, such as the X-Windows system or Mosaic.

Our initial experiments have focused on Map-CSP and a simple variant of Map-CSP that uses pre-filtering of domain values to more efficiently recognize program plans. However, Map-CSP uses FCDR, a straightforward approach to solving CSP problems that relies on the size of domain value sets to determine which constraints to consider. FCDR is never optimal, although always good (Kondrak and van Beek, 1995). As a result, we need to experiment with other approaches, such as those that rely on knowledge about the relative effectiveness of different constraints and those that group domain value sets to speed the CSP solution process. In particular, data-flow constraints appear to be more powerful than control-flow constraints in eliminating domain values, and components can be organized into groups depending on which specific variables they access.

Our initial experiments have explored the scaling properties of our CSP-based understanding algorithm in terms of program size (by using plans of a given size with different size programs). We have not, however, explored the scaling properties of this algorithm in terms of plan size (number of components and constraints). While program understanding as been shown to be worst-case exponential in terms of plan size (Woods and Yang, 1996), there is reason to believe that large plans may not be significantly harder to locate than small plans, as a result of the effectiveness of data flow constraints at narrowing down the search space. One way to address this question is to search our existing test programs for plans of varying size and complexity.

Our initial experiments have also focused on only a single part of the understanding problem: verification that a plan is present. We haven't experimented with H-Map-CSP as a whole to determine the total cost to understand a non-trivial program, nor have we searched for a wide range of different plans within a single program. As a result, it's necessary for us to perform experiments using H-Map-CSP on real-world programs with a real-world plan library. Our likely experiment will be to take a hierarchical library of "Year 2000" plans and a set of real-world COBOL programs and explore the performance of the algorithm in recognizing these plans. This experiment is particularly attractive not only because it addresses an important real-world problem, but also because our initial study has indicated that many Year 2000-related plans (e.g., detecting a leap year) tend to be fairly small, along the same sizes of plans for which we have already searched. As a result, such an experiment may lead to a practical confirmation of our previous performance results.

Finally, we need to perform experiments that compare the performance of our approach to program plan recognition with existing special-purpose algorithms for program plan recognition. We have done some initial work in this direction, which involves mapping other approaches, such as the Concept Recognizer (Kozaczynski and Ning, 1994) and DECODE (Chin and Quilici, 1996; Quilici, 1994) into a CSP framework. This initial work shows that the CSP approach compares favorably with at least several of these existing algorithms (Quilici and Woods, 1997). There are, however, a wide variety of different program understanding algorithms that we haven't yet explored, such as those used by GRASPR (Wills, 1992; Wills, 1990). In addition, we have also now found a more effective constraint evaluation algorithm, which suggests that we should redo these earlier efforts.

6.2. Alternatives to H-Map-CSP

H-Map-CSP is just one approach to program understanding, and it may not be the most efficient way to make use of CSP techniques. An alternative would be to view understanding the entire program as a constraint satisfaction problem, not just recognizing instances of a single, hypothesized plan (Woods and Yang, 1995).

In this model, called the *program-understanding CSP*, or PU-CSP, a program is first divided into blocks. Each block is a set of closely related source code. The program understanding problem is then to identify the top-level function of each of these program blocks, so that we not only explain the inter-relationships among the blocks but also respect the constraints specified by a program library on the program plans describing the block.

Rather than treating understanding as a sequence of independent problems, we can instead represent this problem as a hierarchical CSP. At one level, we are worried about the inter-relationships between explanations for blocks. Each block is represented as a variable in a CSP and the plan components that can be used to explain the block give rise to the values for that variable. The data flow and control flow relationships between block may be seen as constraints among these variables. At a lower level, we worry about determining the possible explanations for these blocks, which we can do with Map-CSP. A hierarchically organized CSP intertwines the process of solving these problems so that reducing the possible solutions for one block has the potential to eliminate solutions for other blocks. A solution to this hierarchical CSP is an overall explanation for a program.

We have constructed this plan recognizer and are beginning to run experiments that compare its performance with the H-Map-CSP approach.

6.3. Imprecise Understanding

While the focus of our paper has been precise understanding, there is also a need for imprecise understanding. Maintenance programmers, for example, are aided by having *guesses* as to what design elements are most likely to be present in the code, even if those elements can not be proven to be there. Ideally, our initial precise understanding can serve as the basis for an attempt to infer higher-level understanding, as shown in Figure 17.

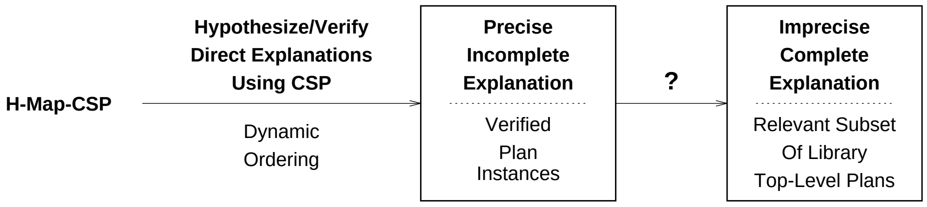
H-Map-CSP
Hypothesize/Verify Direct Explanations Using CSP
Dynamic Ordering
Precise Incomplete Explanation
Verified Plan Instances
?
Imprecise Complete Explanation
Relevant Subset Of Library Top-Level Plans

Figure 17. The architecture of an extension to our current approach.

One approach to hypothesizing higher-level plans is to attempt to run through those plans for which we failed to find any instances, heuristically relax the constraints on those plans, and attempt to re-run Map-CSP on each. It's possible that the relaxed Map-CSP will find new plans, which we are then inferring to be present in the code. This execution of relaxed Map-CSP can be done in layers, as in H-Map-CSP. The ideal end result is a set of top-level plans that explain the original actions, but not with certainty, as not all of their constraints were tested and satisfied. The key issue in this approach is in determining which ways to relax a particular plan out of the many possible ways to do so.

We are planning to flush out and explore this relaxed-CSP approach as a mechanism for integrating precise and imprecise program understanding.

7. Relevance To The Real-World

We have focused primarily on the scalability of algorithms for recognizing program plans. But suppose it turns out that recognizing instances of a given plan is sufficiently tractable that we can deploy our understander and apply it to real-world programs—then what?

One key problem is that the plan recognizer requires a library of program plans. Our simple example to illustrate the behavior of Kautz's algorithm showed that a relatively complex hierarchy is required to understand just a few lines of code. That implies that a significantly more complex hierarchy will be required to understand 10,000-line modules. It's clear that to apply plan-based understanding to real-world systems we will need a cost-effective way to create plan hierarchies.

We have begun exploring an approach for helping programmers construct a plan library. The idea is to provide programmers with a tool that allows them to provide plans by example. In particular, the approach is to let them highlight existing code as an instance of a plan, provide them with a detailed view of the components and constraints present in this instance, and allow them to delete and/or generalize constraints and components. The system can support this process by checking whether various combinations of the components/constraints present in this plan instance correspond to already-entered plans, and then automatically grouping and replacing them with previously-defined plans. The end result is a definition of the plan and links from it to other library entries. Given a sufficiently fast program understanding algorithm, the set of programs that may contain the user-provided plan can be immediately searched, and the user can adjust the plan's definition based on the results.

Besides the technical issues involved in constructing this tool, it's an open, empirical question whether such a tool can be used to cost-effectively provide plan libraries. However, it does suggest one possible path toward addressing the problem of how the necessary plans are provided to program understanding systems. Such a tool also suggests one near-term application of plan-based program understanding technology: letting users locate *conceptually* similar code fragments within a set of source files by using the tool to specify characteristics of the code to search for.

8. Conclusion

Program understanding is often viewed as a task of understanding the plans inherent in a piece of source code. We have demonstrated that there are serious problems with the naive notion of directly applying plan recognition algorithms and that these problems in some sense justify the rejection of these algorithms by researchers in program understanding. In particular, we have shown that simply applying AI plan recognition algorithms to program understanding is not only inefficient but can also lead to incorrect results.

We have discovered that generalized plan recognition and program plan recognition differ in terms of a key underlying assumption. While general plan recognition assumes "open perception" (not all actions are known), program understanding assumes "closed perception" (all actions are known), and we have shown how this difference allows program plan recognition algorithms to more efficiently recognize plans. In particular, it allows program plan recognition algorithms to process actions in an arbitrary rather than fixed order and to eliminate quickly many hypothesized explanations from further consideration, rather than carrying around a potentially large collection of unverified hypotheses.

We have provided a constraint-based program plan recognition algorithm that takes advantage of these improvements, and we have provided some initial empirical evidence that this algorithm appears to efficiently recognize certain classes of plans in real-world programs—despite program understanding having been shown to be NP-hard (Woods and Yang, 1996). In particular, our experimental results strongly suggest that program plan recognition is potentially quite tractable for programs of up to 10,000 lines with plans of a similar size and complexity to the ones we have tried.[9] While many real world programs are far larger, there exist techniques that can be used to semi-automatically divide them into collections of modules in this size (Newcomb and Markosian, 1993). This suggests that we are nearly at the point where we can begin to apply plan-based program understanding techniques to real-world legacy systems. In addition, there are many real-world *modules* of 10,000 lines or less, and it appears to be worth trying to apply our CSP-based plan recognition techniques to those modules.

Finally, we believe we have made a contribution not only to the world of program understanding but also to AI in general. In particular, our plan recognition algorithm may be applicable to any other plan recognition problem in which the closed perception assumption holds. One obvious place where this is the case is the problem of recognizing high-level user plans from detailed audit logs of user actions, and there are likely to be others. This suggests not only that program understanding is an interesting domain for further study by AI researchers but also that there is the potential for program understanding results to apply more generally.

Acknowledgments

The authors would like to acknowledge the US Air Force Rome Labs, the National Science Foundation, the Natural Sciences and Engineering Research Council of Canada, and the Information Technology Research Centre (ITRC) for their support. Qiang Yang would also like to thank the following for support: MPR Teltech Ltd., Science Council of BC, BC Advanced Systems Institute, The Simon Fraser University, Ebco Industries Ltd., Epic Data International, MacDonald Dettwiler and Associates, Glenayre Electronics Inc., BC Telephone Company and Ontario Centers of Excellence (ITRC).

We also thank Yongjun Zhang, who implemented data and control-flow analysis tools for a subset of C, and who set up and ran many of our experiments, as well as Prem Devanbu and AT&T, who allowed us the free use of Gen++ to help us construct C program analyzers.

Finally, we thank Stephanie Ellis, Robin Cohen, and the KBSE ’96 reviewers for useful comments on earlier drafts of this work.

Notes

1. By plan, we mean a particular code pattern. There may be more than one way to perform a particular task with each way represented by a unique code pattern.
2. *Best* is a highly subjective term which changes definition depending on the intent of the particular plan recognition application.
3. For simplicity, and without losing generality, we focus on the non-dichronic variant of the Kautz and Allen recognition algorithm. This version obtains the same results regardless of the order in which observations appear.
4. The Kautz algorithm does not throw away individual explanation chains once they have been merged with other changes.
5. See (Kumar, 1992) for an accessible and detailed treatment of Constraint Satisfaction Problems.
6. The specific plan we search for is about the same size and complexity as in our original set of experiments, but the details vary due both to the different constraints available in real-world programs and to differences in the specific components used to represent the actions in these programs.
7. The cost of this step should be a constant times the total number of domain values for all plan components.
8. For this experiment, implemented in Lisp on a Sparc1000 workstation without any attempt at serious optimization, it took less than 30 seconds of CPU time to recognize all instances of a particular plan in our 5,000 line programs and less than 90 seconds of CPU time in our largest 10,000 line program.
9. We must keep in mind that the search for plan instances is exponential in the size of the plans, and consequently these results will not hold in general for larger or less well-constrained plans.

References

Chin, D. and Quilici, A. 1996. DECODE: A cooperative program understanding environment. *Journal of Software Maintenance*, 8:3–34.

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

Freuder, E. and Wallace, J. 1992. Partial constraint satisfaction. *Artificial Intelligence*, 58: 21–70.

Hartman, J. 1991a. Automatic control understanding for natural programs. PhD thesis, University of Texas at Austin, Department of Computer Science.

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

Kautz, H. 1987. A formal theory of plan recognition. PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York.

Kautz, H. and Allen, J. 1986. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, pp. 32–37.

Kondrak, G. and van Beek, P. 1995. A theoretical evaluation of selected backtracking algorithms. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, CA, pp. 541–547.

Kozaczynski, W. and Ning, J. 1994. Automated program understanding by concept recognition. *Automated Software Engineering*, 1: 61–78.

Kumar, V. 1992. Algorithms for constraint-satisfaction problems. *AI Magazine*, 13: 32–44.

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

Mackworth, A.K., Muldter, J. and Havens, W. 1985. Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. *Computation Intelligence*, 1 :188–196.

Minton, S., Johnston, M., Philips, A. and 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.

Newcomb, P. and Markosian, L. 1993. Automating the modularization of large COBOL programs: application of an enabling technology for reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, pp. 222–230.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9: 268–299.

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

Quilici, A. 1995. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, pp. 333–336.

Quilici, A. and Woods, S. 1997. Toward a constraint-satisfaction framework for program understanding. *Journal of Automated Software Engineering*, 4: 271–289

Sidebottom, G. and Havens, W.S. 1992. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8: 601–623.

Sosic, R. and Gu, J. 1990. A polynomial time algorithm for the n-queens problem. *SIGART*, 1.

Song, F. and Cohen, R. 1991. Temporal teasoning during plan recognition, In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, pp. 247–252.

van Beek, P., Cohen, R., and Schmidt, K. 1994. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence*, 9: 132–154.

Wills, L. M. 1990. Automated program recognition: a feasibility demonstration. *Artificial Intelligence*, 45: 113–172.

Wills, L. M. 1992. Automated program recognition by Graph Parsing. PhD thesis, MIT, Department of Computer Science.

Woods, S. 1996. A method of program understanding using constraint satisfaction for software reverse engineering. Ph.D. Thesis, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Woods, S. and Yang, Q. 1996. The program understanding problem: Analysis and a heuristic approach In *Proceedings of the 18th International Conference on Software Engineering ICSE-96*, Berlin, Germany, pp. 6–15.

Woods, S. and Yang, Q. 1995. Program understanding as constraint satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering*, Toronto, Ontario, Canada, pp. 318–327.

Yang, Q. and Fong, P. 1992. Solving partial constraint satisfaction problems using local search and abstraction. Technical Report CS-92-50, University of Waterloo, Waterloo, Ontario, Canada.

Zhang, Y. 1997. Scalability experiments in applying constraint-based program understanding algorithms to real-world programs. Masters Thesis, Department of Electrical Engineering, University of Hawaii at Manoa, Honolulu, Hawaii.