A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms *

Alex Quilici
Dept. of Electrical Engineering
University of Hawaii
2540 Dole St., Holmes 483
Honolulu, HI 96822
alex@wiliki.eng.hawaii.edu

Steven Woods
Dept. of Computer Science
University of Waterloo
Davis Centre for Computer Research
Waterloo, ON N2L 3G1
sgwoods@logos.uwaterloo.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.

Introduction

Over the past decade, researchers have proposed and implemented a wide variety of plan-based program understanding algorithms [14, 7, 8, 17, 18, 4, 5]. 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[17, 1, 20], 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 [20, 21], 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[6] and local search-based methods[13, 10, 22], constraint-

*This work was partially supported by the KBSA project, Air Force Rome Labs, under Air Force contract #F30602-93-C-0257, the Natural Sciences and Engineering Research Council of Canada, and the Institute for Computer Research (ICR). The authorship of this paper is alphabetical.

[1]See [9] for an overview of constraint satisfaction.

propagation problem simplifications[11, 2, 12], hierarchical exploitation of problem structure[3], 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 An Existing Plan-Based Program Understanding Algorithm

This section describes an existing plan-based program understanding algorithm [14]. This algorithm was derived from studies of users doing bottom-up understanding on functions in C code [15] and used in a cooperative program understanding environment [1]. 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 [7, 8]. 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 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 [7]. 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 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

```
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: An example code pattern.

approaches). Evaluating the index involves checking 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

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 evaluating C_i on each item in $NPPI$.
  - Set PPI to the concatenation of all the $RPPI$s.

Figure 2: Our algorithm for automatically recognizing plan instances in code.

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 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.

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 [20]).

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[16]. 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 [1]. 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 [19] 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.

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) 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 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).

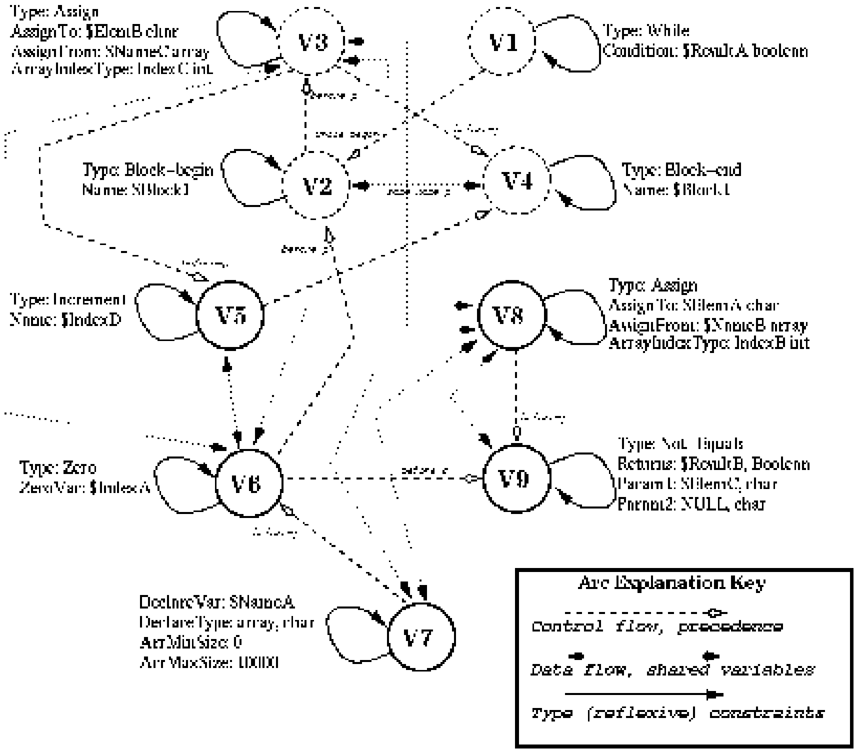

Figure 3: An example plan in the MAP-CSP representation.

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 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

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

[3] See [3, 22] for a detailed discussion of hierarchical CSPs and their solutions.

```
(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 4: MAP-CSP representation of code patterns.

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 4 is an example showing how the portion of the plan of Figure 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) 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 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 predetermined 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.

4 Experimental Discussion

Earlier work[20, 21] 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 with 9 components and 20 constraints has shown even more promising results. Figure 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.

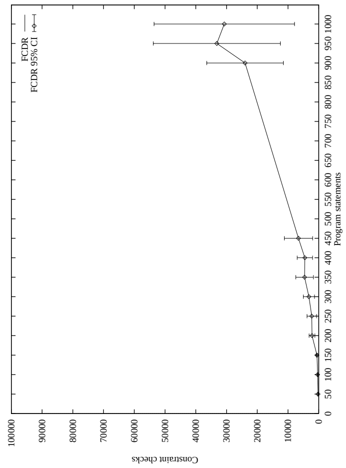

Figure 5: Forward Checking, DR (95% conf. interval).

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

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

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.

References

[1] Chin, D. and Quilici, A. DECODE: A Cooperative Program Understanding Environment. *Journal of Software Maintenance*, to appear.

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

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

[4] Hartman, J. Understanding Natural Programs using Proper Decomposition. In *Proceedings of the International Conference on Software Engineering.* Austin TX, pp. 62-73, 1991.

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

[6] Kondrak, G. and Van Beek, P. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 541–547, 1995.

[7] Kozaczynski, V.; and Ning, J. Q. Automated Program Understanding By Concept Recognition, *Automated Software Engineering 1*, 1 (March 1994), 61–78.

[8] Kozaczynski, V.; Ning, J. Q.; and Engberts, A. Program Concept Recognition and Transformation. *Transactions on Software Engineering 18*, 12 (December 1992), 1065–1075.

[9] Kumar, V. Algorithms for Constraint-Satisfaction Problems. *AI Magazine* (Spring 1992), 32–44.

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

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

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

[13] Sosic, R. and Gu, J. A polynomial time algorithm for the n-queens problem. *SIGART*, 1(3), 1990.

[14] Quilici, A. A Memory-Based Approach to Recognizing Programming Plans. *Communications of the ACM 37*, 5 (May 1994), 84–93.

[15] Quilici, A. "A Hybrid Approach to Recognizing Programming Plans." In *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, May, 1993, 126-133.

[16] Von Mayhrhauser, A. and Vans, A. M. Program comprehension during software maintenance and evolution. *IEEE Computer*, 1995, 44–55.

[17] Wills, L. M. *Automated Program Recognition by Graph Parsing.* Ph.D. Thesis, Technical Report 1358, MIT Artificial Intelligence Lab, Cambridge MA, 1992.

[18] Wills, L. M. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence 45*, 1-2 (1990), 113–172.

[19] Woods, S. A method of interactive recognition of spatially defined model deployment templates using abstraction. In *Proceedings of the Knowledge Based Systems and Robotics Workshop*, pages 665–675. Government of Canada, November 1993.

[20] Woods, S. and Yang, Q. Program Understanding As Constraint Satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, Toronto, CA, 318–327. (Also appears in *Proceedings of the Second Working Conference on Reverse Engineering (WCRE-95)*, July 1995, Toronto, CA.

[21] Woods, S. and Yang, Q. Constraint-based plan recognition in legacy code. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE-95)*, August 1995 (In conjunction with IJCAI-95).

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