, , 1–20 ()

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

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

STEVEN WOODS sgwoods@logos.uwaterloo.ca
Department of Computer Science, University of Waterloo, Davis Centre for Computer Research, Waterloo, ON N2L 3G1

Received January 4, 1996 ; Revised March 13, 1996

Editor: Chris Welty

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 complex existing program understanding algorithm into an instance of a constraint satisfaction problems, and showing this facilitates better understanding of its performance.

Keywords: Program understanding, constraint satisfaction, performance evaluation

1. Introduction

Over the past decade, researchers have proposed and implemented a wide variety of plan-based program understanding algorithms (Quilici, 1994; Kozaczynski et al., 1992; Wills, 1992, Wills 1990; Hartman, 1991; Johnson, 1986). Although the problem has been shown to be NP-hard (Woods and Yang, 1996), some of these research efforts have presented promising empirical results in mapping plan libraries to reasonably sized (up to 1000 lines) legacy source code (Wills, 1992; Wills, 1990; Chin and Quilici, 1996; Woods and Yang, 1995a). None, however, 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 (Wills, 1992; Wills, 1990),

* 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 Information Technology Research Centre. The authorship of this paper is alphabetical.

components and constraints (Quilici, 1994; Kozaczynski and Ning, 1994; Kozaczynski et al., 1992), regular expressions and transformation rules (Johnson, 1986), and so on, and to use collections of heuristic tricks to improve performance, such as indexing (Quilici, 1994), specialized rule and constraint ordering (Kozaczynski and Ning, 1994; Kozaczynski et al., 1992; Wills, 1990), 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. Earlier work (Woods and Yang, 1995a; Woods and Yang, 1995b) proposed viewing program understanding as a *constraint satisfaction* problem (CSP)[1] and demonstrated how a CSP-based approach could successfully address one portion of the program understanding problem (mapping program plans directly to program source code). As a result, it is natural to wonder whether other, existing program understanding algorithms, despite their differing representations and heuristic tricks, can also be mapped into this CSP-based framework. If this framework is or can be made 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 include global (Kondrak and Van Beek, 1995) and local search-based methods (Sosic and Gu, 1990; Minton, 1992; Yang and Fong, 1992), constraint-propagation problem simplifications (Nadel 1989; Dechter 1992; Prosser 1993), hierarchical exploitation of problem structure (Freuder and Wallace, 1992), as well as hybrid combinations of these approaches.

This paper provides a constraint satisfaction framework and demonstrates how one well-known heuristic program understanding algorithm can be placed within that framework, how this improves our understanding of its behavior and performance, and how this viewpoint facilitates comparing its performance with other program understanding algorithms. In particular:

- Section 2 provides an initial representation for program plans and program understanding algorithm, along with a constraint satisfaction framework that addresses it.

- Section 3 describes an existing extension to that plan representation and algorithm, and shows how it can be addressed within the constraint satisfaction framework, while preserving both its representational framework and heuristic tricks.

- Section 4 presents a detailed example of our implementation of the extended algorithm in this constraint-satisfaction based framework.

- Section 5 provides a comparison in performance between the constraint satisfaction approach and the original approach.
- Section 6 summarizes our future research path and the conclusions we've drawn from our current work.

2. Program Understanding as a CSP

Program understanding involves recognizing instances of program plans from source code. This involves representing program plans and then providing an algorithm for hierarchially matching those plans against the source code. This section describes a straightforward approach to this task and shows how it can be modeled in a CSP-based framework.

2.1. A Straightforward Approach to Program Understanding

One way to represent program plans, originally used in the Concept Recognizer (Kozaczynski and Ning, 1994; Kozaczynski et al., 1992), is as a combination of attributes (which are instantiated when a plan instance is recognized) and a set of common implementation patterns, where each code pattern is a collection 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 be hold between these components).

Figure 1 contains an example of a simple representation of a plan.

```
define TRAVERSE-STRING(String) isa TRAVERSE-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)
```

Figure 1. An example code pattern.

This figure shows how the plan TRAVERSE-STRING is represented, where this plan captures the common notion of traversing each character in a C string). The components are syntax tree entries and subplans. In particular the components are a DECL-ARRAY to declare the character array, a ZERO sub-plan to initialize the index variable to zero, a LOOP, two ACCESSes 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.

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.

2.2. An Initial CSP Framework

How can we place the Concept Recognizer's program understanding approach in a CSP 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.

A key aspect of the program understanding problem is finding all instances of a particular plan in a program's internal representation (which includes the program's AST, along with any previously recognized plans). We can address this problem as a CSP, called MAP-CSP, in the following way.

MAP-CSP models each plan component as a variable. Each variable has a domain ranging over the actual AST entries or recognized sub-plans that satisfy a set of constraints on the "type" of the variable, and the actual occurrences of each of these components in the source code correspond to possible domain values for the variables. These "type" constraints are reflexive, in that they affect one variable only. They are derived from the partial naming and typing information provided in the component description. For example, the component 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 is of type character of any size or any name. However, it is easy to imagine components that would map into more tightly constrained CSP variables.

MAP-CSP models the constraints among plan components (such as the various data-flow and control-flow relationships that must hold between them) as inter-variable constraints between plan variables. For instance, in the example plan of Figure 1, there is a constraint `ControlPath` that exists between the `DECL-ARRAY` and the `LOOP`, such that the `DECL-ARRAY` logically precedes the `LOOP`. This is directly mapped to the CSP representation, where any instance of the variable corresponding to `DECL-ARRAY` is contrained to logically precede any instance of the variable corresponding to the `LOOP` component.

Figure 2 details the variables and constraints of the resulting MAP-CSP for our example plan.

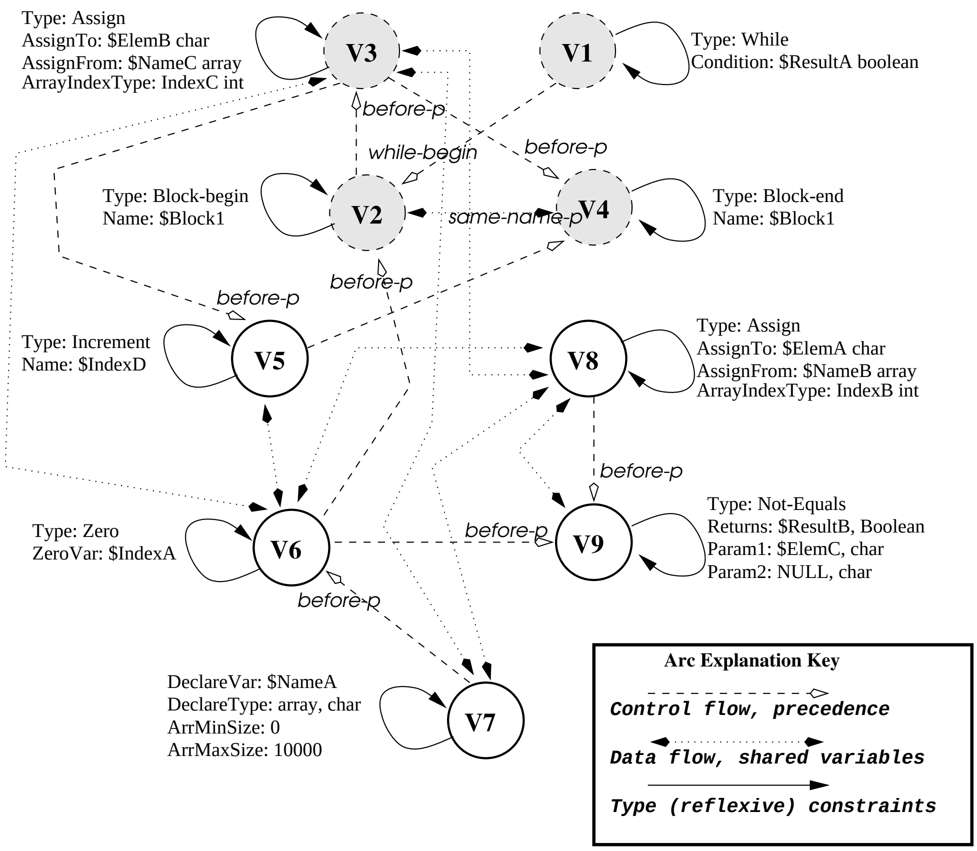

Figure 2. An example plan in the MAP-CSP representation.

It is important to note that this transformation to a CSP is representational only. A particular method of *solving* a CSP requires a mapping of previous control strategies to specific CSP solution approaches.

The result is that a solution to the MAP-CSP is any assignment of domain values (AST elements, or previously recognized plans) to plan variables (plan parts) that satisfies the constraints among the variables (data-flow and control-flow relationships), and corresponds to an instance of a plan that we have identified.

A single MAP-CSP application corresponds to searching for all the instances of a given plan. We can find all instances of all plans present in the source by repeated applications of MAP-CSP. In particular, we can divide the plan library up into layers, where the plans at each level are constructed only from plans at lower levels. That is, the bottom layer is those plans whose components are all AST items, the next layer is plans whose components are a combination of AST items and plans in the bottom layer, and so on. For example, 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. We then run through the plan library "bottom up", by invoking MAP-CSP for each plan in the bottom layer, then each plan in the next layer, and so on. 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 the MAP-CSPs at the lower layers to locate the possible domain values for the components at the higher levels. The overall result is that we deal with hierarchical plan structure through a layered plan library and applications of MAP-CSP a layer at a time.

In general, there are two primary concerns when trying to model a particular program understanding methodology in a constraint-based framework: representation and control. We must ensure that the CSP reprsentation 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.

For this case, the CSP representation captures exactly the original component and constraint representation of plans. And the CSP control strategy is similar to that of the original Concept Recognizer, except that it imposes a particular layered ordering in how plans are tried from the library, where this was unspecified in the original Concept Recognizer descriptions.

3. Heuristic Program Understanding as a CSP

We have demonstrated how to take the Concept Recognizer's approach and turn it into a constraint satisfaction problem. However, although the Concept Recognizer's representation of plans and top-down algorithm for recognizing them is simple and clear and has been successfully applied to real-world COBOL programs, the algorithm is slow and does not scale well, either with program size or plan library size (Kozaczynski and Ning, 1994).

Later work, in a system called DECODE (Chin and Quilici, 1996; Chin, 1994), tried to address these deficiencies by modifying the Concept Recognizer's initial representation and algorithm to reflect the behavior observed from studies of users doing bottom-up understanding of function in C code (Quilici, 1993). This extended algorithm had two key changes. First, DECODE's algorithm became 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, DECODE's algorithm 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. These heuristic tricks are designed to make it more efficient and to help it better model the observed user behavior. The question is, can those heuristic tricks be captured in a CSP-based framework?

3.1. DECODE's Heuristic Approach to Program Understanding

In this section, we describe DECODE's more complex algorithm in some detail and then show how it too can be modelled as a constraint satisfaction problem.

3.1.1. Representation

Figure 3 contains several examples of DECODE's extended plan representation.

```
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 3. An example code pattern.

As in the Concept Recognizer, each plan consists of a set of components and constraints. However, each plan also has an index that says when it should be considered (that is, fully matched against known program pieces and recognized plans). The index combines a plan component with one or more plan constraints and suggests that the plan should be considered whenever this component is encountered and the specified constraints hold. `TRAVERSE-STRING`, for example, is indexed by an `ACCESS` that is contained within a `LOOP`. That means the understander considers this plan each time it encounters an `ACCESS`, not every time it encounters any `INCREMENT`, `ZERO`, `BIN-OP`, `LOOP`, or `DCL-ARRAY` (as in most bottom-up approaches). Evaluating the index involves checking 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 motivation for indexes is that they 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, DECODE's 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 motivation underlying 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, DECODE's representation allows plans to be defined as *specializations* of other plans; that is, as a set of constraints on an existing plan's attributes. For example, the plan ZERO is defined as a specialization of an ASSIGN whose Source is 0. These specializations correspond to plans that contain a single component (the plan being specialized), that are indexed by that component, and that have constraints on that component's attributes. In fact, at definition time, these specializations are automatically translated into standard plan definitions.

The motivation for 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.

3.1.2. Control

Figure 4 shows the algorithm used by DECODE. 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, DECODE's algorithm assumes that the plan library has been organized in layers, just as in our CSP-version of the Concept Recognizer. 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

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 4. DECODE's algorithm for automatically recognizing plan instances in code.

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. DECODE's alternative is to have an ordering for constraints and to form combinations only as they become necessary to evaluate these constraints.

3.2. DECODE's Approach to Program Understanding as a CSP

We can capture DECODE's approach to program understanding with several extensions to our CSP-based framework. The additional parts of DECODE's plan representation we must map to the CSP methodology are the `INDEX` and `IMPLICATION` entries of a plan. This is done through further specifying MAP-CSP's search control strategy.

DECODE's algorithm traverses the program source (or, more precisely, it traverses the abstract syntax tree) 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, we can model this behavior by having the performed MAP-CSP utilize 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. Essentially, then, there are two phases. The first is the index phase in which the indexed portion of the plan is matched in the source code, giving essentially a list of areas of focus. The second or resolution phase is the attempted resolution of each of the index hits into full blown plan instances. We refer to the result CSP mapping the index behaviour of DECODE as the *Memory-CSP*.

This separation into two phases is important, as illustrated by the following example. We define a template with 5 variables. A and B are the index part (say a reference inside a loop), while C, D and E are the remainder (assume there are constraints as well). Memory-CSP should first find all solutions to the partial CSP involving only A and B first. These index hits should be viewed as a set of *indepedent* partial solutions. Now, each of these partial solutions should be refined in turn, returning only those that fit as part of whole solutions involving C, D and E. Effectively, the second phase involves CSPs of only 3 variables each.

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

We handle implication in a similar way to indexing. Any plan that is implied by another can be thought of as being indexed by the plan and any of the implication constraints. As a result, when we process a plan library layer, 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.

In addition, as we run through each layer of MAP-CSPs, 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.

4. 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 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 5 is an example showing how the portion of the plan of Figure 3 corresponding to the index is actually represented.

```
(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 5. MAP-CSP representation of code patterns.

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 3) 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 3 as a combination of a variable `v1` of type `While`, a variable `v2` of type `Begin`, and a variable `v3` of type `End`. We represent the program plan index constraint that the `Assign` exist inside the control environment of the `While` with the pair of precedence constraints placing `v3` after the `v2` instances and before the `v4` instances.

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

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

5. Some Initial Experiments

We have run several experiments to compare the performance of DECODE's heuristic program understanding algorithm against various techniques for solving constraint-satisfaction problems. These experiments focused on exploring the scalability of the various approaches in the problem of locating all of the instances of a particular plan. Our overall approach was to implement a general constraint satisfaction-based framework and then place DECODE's algorithm within that framework.

5.1. Experimental Description

Our prime interest is the performance comparison of different approaches to program understanding in terms of the size of the programs being understood. In particular, our focus is comparing the time taken to recognize all instances of a single plan as programs increased in size. To keep the focus on scale issues alone, our desire was to have programs of varying sizes available where those programs contain correspondingly more instances of the plan as the programs increase in size, and to have programs with the same relative distribution of different program entities (the same percentage of loops, etc...) regardless of size. As a result, automatically we generated test programs to be understood, starting with an instance of a given plan, adding program statements surrounding each instance and adding more instances until we constructed programs of the desired sizes. In particular, we added statements randomly according to a distribution that corresponded to our observed distribution against a cross-section of student C programs.

Figure 6 shows our internal representation of our earlier example plan. This preserves the basic component and constraint representation, although the specific constraints vary from those used in the original systems. In particular, we approximate control and data-flow constraints using locality and containment constraints. In addition, we require a `same-name-p` constraint to capture the notion that a variable appeared in multiple places represents the same underlying entity, a notion that is implicit in DECODE's representation for plans.

```
'( "quilici-t1"
   (
     (q1-c While       (ResultA (boolean)))
     (q1-d Begin       (Block1 (block)))
     (q1-g Assign      (NameC (array (char))) (IndexC (int))
                           (ElemB (char)))
     (q1-e End         (Block2 (block)))
     (q1-i Increment   (IndexD (int)))
     (q1-a Decl        (NameA (array (char) (0 10000))))
     (q1-b Zero        (IndexA (int)))
     (q1-f Assign      (NameB (array (char))) (IndexB (int))
                             (ElemA (char)) )
     (q1-h Not-Equals (ElemC (char)) (NULL (char)) (ResultB (boolean)))
   )
   (
     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
     (before-p    (q1-b q1-c))
     (before-p    (q1-a q1-b))
     (before-p    (q1-b q1-h))
     (before-p    (q1-d q1-e))
     (before-p    (q1-f q1-h))
     (before-p    (q1-g q1-i))
     (before-p    (q1-d q1-i))
     (before-p    (q1-i q1-e))
     (same-name-p (q1-c q1-h) (ResultA ResultB))
     (same-name-p (q1-f q1-h) (ElemA ElemC))
     (same-name-p (q1-a q1-f) (NameA NameB))
     (same-name-p (q1-a q1-g) (NameA NameC))
     (same-name-p (q1-b q1-f) (IndexA IndexB))
     (same-name-p (q1-b q1-g) (IndexA IndexC))
     (same-name-p (q1-b q1-i) (IndexA IndexD))
  ))
```

Figure 6. Our actual internal representation for plans.

Given this experimental framework, we generated programs of varying sizes at intervals of 50 from 50 to 1000 added lines, with 10 programs at each size. Based on these 10 data points at each size level, we generate a 95% confidence interval for the number of constraint checks occuring during the search. The idea is that constraint checks are a reasonable measure of relative performance, since across our methodologies the work peformed for each constraint check is consistent. While certain methodologies require differing amounts of computation during the search, we have verified that the CPU-second graphs of these same experimental results yields comparable graphs. For comparison, examples requiring approximately 2500 constraint checks utilize roughly 6 CPU seconds in our examples. Since CPU usage is highly variable across implementations and platforms, constraint checks offer a more domain-independent reference point.

5.2. Methodologies Tested

Our primary goal is to model DECODE's heuristic approach (which we refer to below as "Memory-CSP") as a particular type of constraint-satisfaction problem and compare its performance to various techniques for solving constraint satisfaction problems. In particular, we tried two variations of Memory-CSP and compared those approaches to two domain-independent heuristic approaches to solving constraint satisfaction problems. As a point of comparison, we also generated naive solutions using simple backtracking for as many problem instances as praticable.

5.2.1. MAP-CSP

We tried three variants of vanilla MAP-CSP (without the ordering suggested by DECODE). The first is based on *Simple Backtracking.* This method provides a basepoint for comparison with the remainder of the experiments. The second and third strategies utilize a well known search strategy known as *Forward Checking with Dynamic Rearrangement* (FCDR). This method works as follows: Say we have four variables A,B,C,D. S(A) is the size of the domain of A, D(A). Say S(A) < S(B) < S(C) < S(D) initially. First, variable domains are ordered by size as shown. Next, the smallest S(v) is chosen. In our example, the variable A is chosen and can thus be seen as the root of the search space. Next, a value in D(A) is chosen, say a1. Based on the value chosen for A, *forward checking* is performed in which the domains of S(B), S(C), S(D) are reduced where any value in these domains inconsistent with a1 according to any constraint applicable between A and B, A and C or A and D - is removed. Next, the variables are *dynamically rearranged.* For instance, since the domains of variables B, C and D are now possibly reduced, they have new sizes, say S1(C) < S1(D) < S1(B). They are reordered according to smallest first and the smallest is selected for instantiation. The process is now repeated until there are no more variables or values to check.

We experimented with two variations on this strategy. In the first, *FCDR without advance variable ordering*, the initial variable selection is random, while in the other, *FCDR with advance variable ordering*, the initial variable selection is based on the smallest domain size.

5.2.2. Memory-CSP

As we saw earlier, DECODE's approach to program understanding can be modelled by a variant of MAP-CSP with two phases: the index phase and the resolution phase. Our implementation models this by dividing the original plans into two parts: the index plan, which leads to a set of solutions which correspond to those possibly indexed plans, and the non-index portion of the plan, which is applied to each indexed-solution. Figure 7 shows the index for our earlier example.

```
'( "quilici-t1-index"
   (
     (q1-c While      (ResultA (boolean)))
     (q1-d Begin      (Block1 (block)))
     (q1-g Assign     (NameC (array (char))) (IndexC (int))
                      (ElemB (char)))
     (q1-e End        (Block2 (block)))
   )
   (
     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
   ))
```

Figure 7. Our representation for a plan index.

Since Memory-CSP has two distinct phases, it is necessary to select search heuristics and parameters for each of the phases. We show two variations here: the first is 2-Phased Memory-CSP with Phase 1 Simple Backtracking and Phase 2 FCDR with advance variable sorting, and the second is 2-Phased Memory-CSP with both phases utilizing FCDR with advance variable sorting.

5.3. Experimental Results and Discussion

There were 5 test cases. Figure 8 shows the results of these tests.

Test case 1 refers to 2-Phased Memory CSP with Phase 1 BT and Phase 2 FCDR with sorting, Test case 2 refers to 2-Phased Memory CSP with Phase 1 and 2 FCDR with sorting. Test case 3 refers to MAP-CSP with FCDR. Test case 4 refers to MAP-CSP with FCDR and advance sorting. Finally test case 5 refers to MAP-CSP with Simple Backtracking.

All experiments are limited by a 600 CPU second time limit for any individual search. For tests 1 through 4, all test cases for 550 or fewer lines of code completed normally. For 600 and more, there were occasional failures due to our approximation of data and control-flow constraints, and increasing numbers of aborted searches due to exceeding the pre-assigned time limit. Test 1 exhibited a majority of successful solutions in noise levels of less than 750, and tests 2, 3, and 4 in noise levels less than 950. Test 5 could not complete in less than 600 CPU seconds for noise levels exceeding 400.

5.4. Summary of Results and Analysis

The bottom-line of these studies is this:

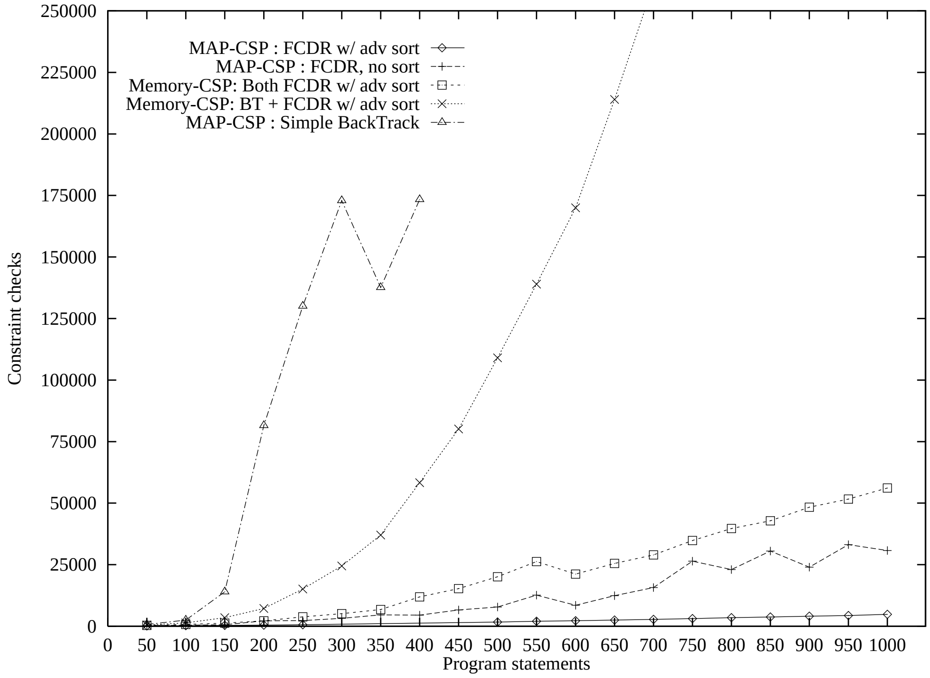

Figure 8. The results of our experiements.

> Heuristic methods for constraint satisfaction other than simple backtracking significantly outperform DECODE's indexed approach.

In hindsight, there is an obvious explanation for why this situation occurs. One is that indexing by its very nature is static—the assumption is that an expert can look at plan, pick its key components, and those components will always be a good filter for whether the rest of that plan should be considered, regardless of the actual entities present in a particular program. By their very nature, the heuristic constraint satisfaction approaches are dynamic, determining which constraints to satisfy based on properties of the particular program being examined.

However, there are some artifacts of our particular experiments that need to be explored futher before our results are validated. One is that our experiments used locality to approximate data and control-flow constraints, and our generated test programs did not necessarily have the same structural properties that real-world programs might have. Since the indexing approach is designed to exploit these structural properties, it may well be that its performance has been arbitrarily limited by our artificial programs. It's now necessary for us to generate programs with real data-flow and control-flow constraints and modify our internal representation to use them, rather than approximations.

It is also possible that our particular approach to mapping heuristic program understanding to constraint satisfaction problems has led to additional constraint matches. The idea is that the two phase approach first finds all N index matches for a given plan and then generates N resolution problems, where we hope that these will either quickly fail or will result in a recognized plan instance. This happens to some extent, but when N is large and the beginning parts of the N resolution search spaces look quite similar (or are nearly identical), a given constraint check between two candidate statements will be checked many (up to N) times. It may well be possible to eliminate these redundant constraint checks by factoring out commonalities between these problems.

6. Conclusions

This paper has demonstrated how to take an existing heuristic-based program understanding algorithm and model it in a constraint-satisfaction based framework. Furthermore, it has presented preliminary results that suggest the domain independent heuristics used by the constraint satisfaction approach lead to significantly better performance than the domain-dependent indexing used in this heuristic-based program understanding algorithm. These results suggest that by casting program understanding as CSP, we can adapt previously developed constraint propogation and search algorithms to improve their efficiency and lead to program understanding algorithms that scale.

One key area of our continuing research is to attempt to verify these initial results. We intend to do so by comparing the performance of our program understanding algorithms with "real-world" programs. Another area is to explore in more detail the specific differences between how our various CSP-based understanders perform plan recognition. Our goal is to verify that we have accurately capture all of the nuances of indexing with our CSP-implementation of that understander and to better understand exactly why the domain-independent CSP algorithm appears to perform so much better. A final area is to explore how to convert other program understanding approaches to our CSP approach so that we can study their performance. We need to determine whether the CSP approach will beat the performance of other algorithms or whether there are other approaches that continue to be promising.

However, even if our preliminary performance results do not pan out, they demonstrate the benefits of having a common framework for comparing the performance of different program understanding algorithms. This framework allows us compare the efficacy of specific heuristic tricks such as indexing to different methods of solving constraint satisfaction problems. It may well prove out in the long run 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 certain constraints) over heuristic constraint propogation methods.

Although we have just begun studying program understanding algorithms in terms of this constraint satisfaction-based approach, it appears promising as a unifying framework for describing and comparing program understanding algorithms. Our hope is that we will be able to extend this constraint-based framework to capture the representations and control strategies used in other approaches to program understanding. The result of doing so should be a deeper understanding of the commonalities and differences of these algorithms. Ultimately, it may also lead us to a deeper understanding of the program understanding problem, and perhaps to a program understanding approach that scales.

Notes

1. See (Kumar, 1992) for an overview of constraint satisfaction.
2. See (Freuder and Wallace, 1992; Yang and Fong, 1992) for a detailed discussion of hierarchical CSPs and their solutions.

References

1. Chin, D., and Quilici, A. 1996. DECODE: A Cooperative Program Understanding Environment. *Journal of Software Maintenance*, 8(1).
2. Dechter, R. 1992. From local to global consistency. *Artificial Intelligence*, 55:87–107.
3. Freuder, E., and Wallace, J. 1992. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70.
4. Hartman, J. 1991. Understanding Natural Programs using Proper Decomposition. *Proceedings of the International Conference on Software Engineering.* Austin TX, pp. 62-73.
5. Johnson, W. L. 1986. *Intention Based Diagnosis of Novice Programming Errors.* Los Altos, CA:Morgan Kaufman.
6. Kondrak, G., and Van Beek, P. 1995. A theoretical evaluation of selected backtracking algorithms. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 541–547.
7. Kozaczynski, V., and Ning, J. Q. 1994. Automated Program Understanding By Concept Recognition, *Automated Software Engineering*, 1(1):61–78.
8. Kozaczynski, V., Ning, J. Q.; and Engberts, A. 1992. Program Concept Recognition and Transformation. *Transactions on Software Engineering*, 18(12):1065-1075.
9. Kumar, V. 1992. Algorithms for Constraint-Satisfaction Problems. *AI Magazine*, 13(1):32–44.
10. 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.
11. Nadel, B. A., Constraint satisfaction algorithms. 1989. *Computational Intelligence*, 5:188–224.
12. Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.
13. Quilici, A. 1994. A Memory-Based Approach to Recognizing Programming Plans. *Communications of the ACM*, 37(5):84–93.
14. Quilici, A. 1993. A Hybrid Approach to Recognizing Programming Plans. *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, pp. 126-133.

15. Sosic, R., and Gu, J. 1990. A polynomial time algorithm for the n-queens problem. *SIGART*, 1(3).
16. Von Mayhrhauser, A. and Vans, A. M. 1995. Program comprehension during software maintenance and evolution. *IEEE Computer*, pp. 44–55.
17. Wills, L. M. 1992. Automated Program Recognition by Graph Parsing. Ph.D. Thesis, MIT Artificial Intelligence Lab, Technical Report 1358, Cambridge, MA.
18. Wills, L. M. 1990. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45(1-2):113–172.
19. Woods, S., and Yang, Q. 1995. Program Understanding As Constraint Satisfaction. *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, Toronto, CA, pp. 318–327.
20. 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.
21. Woods, S., and Yang, Q. 1995. Constraint-based plan recognition in legacy code. *Working Notes of the Third Workshop on AI and Software Engineering: Breaking the Toy Mold (AISE-95).*
22. Yang, Q. and Fong, P. 1992. Solving partial constraint satisfaction problems using local search and abstraction. University of Waterloo, Technical Report CS-92-50, Waterloo, Ontario.

Received Date
Accepted Date
Final Manuscript Date