A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering

Steven Woods

Department of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA

July, 1996

Presentation Overview

I (3-14) Introduction

- → **Software, Software Reverse Engineering**
- → **Program Understanding and Importance**

II (15) Definition of Thesis Sufficiency Conditions

- → Complexity; → CSP Modeling, Coverage;
- → Hierarchical CSP; → Implementation;
- → Empirical Results; → Interactive Integration;

III (16-26) Accomplishment of Sufficiency Conditions

IV (27) Ongoing Work and Future Research Directions

V (28) Contribution Review

Introduction: Software

Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike. Fred Brooks Jr., 1995.

Man with Cuboid
M.C. Escher, 1958

The Nature of Software: Complex

- Software models large, abstract, & complex processes
- Software hard ($) to create, maintain, extend & <u>understand</u>
  - → Special domain knowledge integral to understanding software
  - → Dispersed, local or non-existent knowledge of code & data
  - → Sporadic, old or non-existent documentation and comments
  - → Multi-dimensional representations hard to visualize
  - → Systems frequently embed old, redundant, fragile code
  - → Poorly designed, freq. changed systems often highly fragile

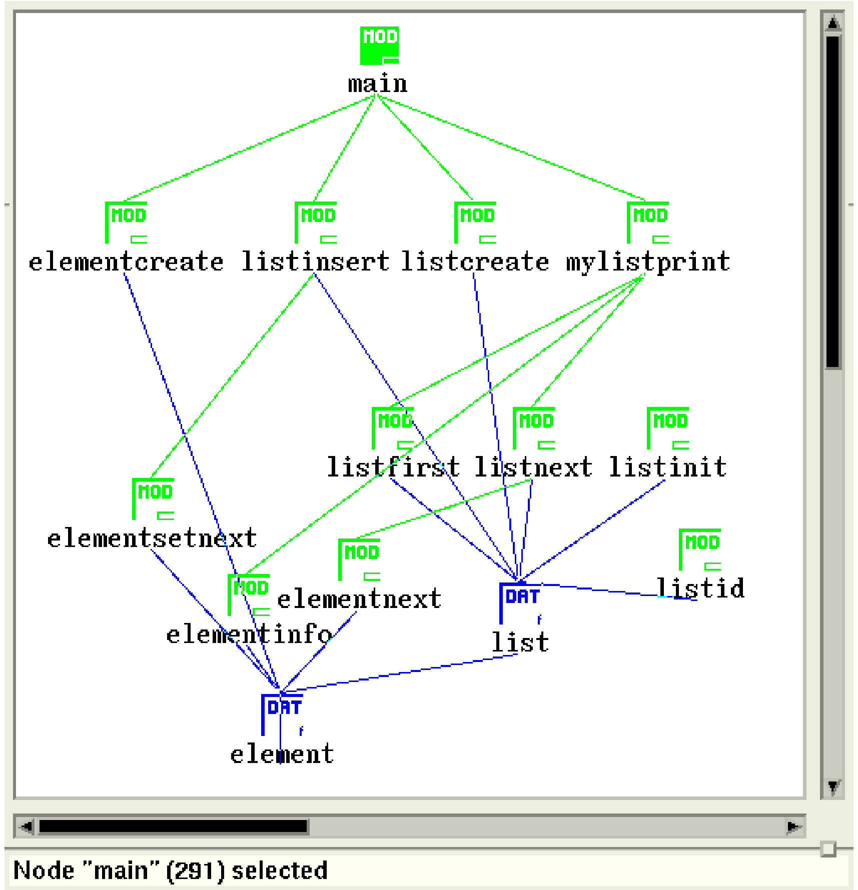
The Nature of Software: Structure

- However, Software has a wealth of *structure*
  - → Data-flow structure
  - → Control-flow structure
  - → Data-use structure
  - → Functional structure
  - → Execution trace structures
- And structure can provide *clues* to purpose

Software Reverse and Re-Engineering

- Reverse Engineers seek to *identify* original system structures
- Re-Engineers seek to *improve* current system structure
- Any software changes require *reverse & re-engineers* who:
  - → Determine system functionality, design from artifacts
  - → Institute changes preserving most functionality
- Complexity handling critical with larger software
  - → Manpower can't overcome knowledge hurdles, ie domains
  - → Critical subtask is *understanding* others' code artifacts
  - → Tool-support critical to cost-effectiveness

Visualization Tools aid Understanding



RIGI displaying parsed artifacts

Reverse Engineering: Knowledge Application

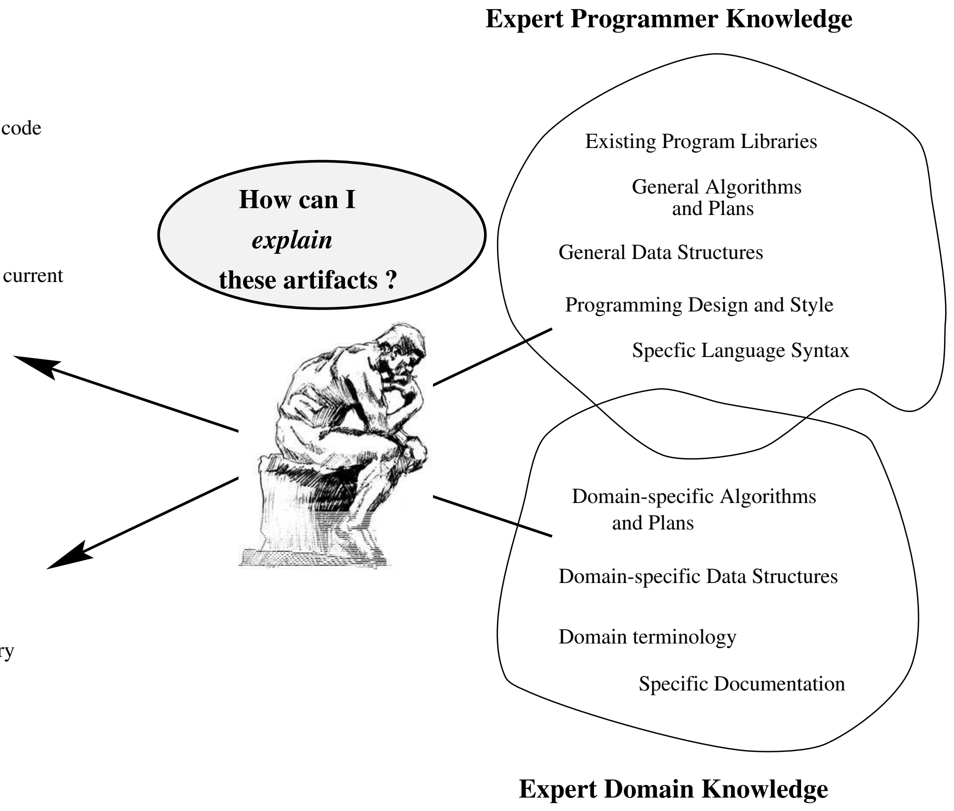
- Software embeds, conceals “programmer plan” instantiations
- Experts must *understand* software in context of:
  - → Known programming plans, structures and methodology
  - → Known relevant application domain knowledge, plans
- To minimize size & complexity, (Re)-Engineers wish to:
  - → Consolidate code functionality if possible
  - → Simplify code via shared/commercial libraries
  - → Rewrite, replace all or some systems

Problem: Software Program Understanding

- Our View: Understanding is a *Knowledge Mapping*

Legacy Source Code Artifacts

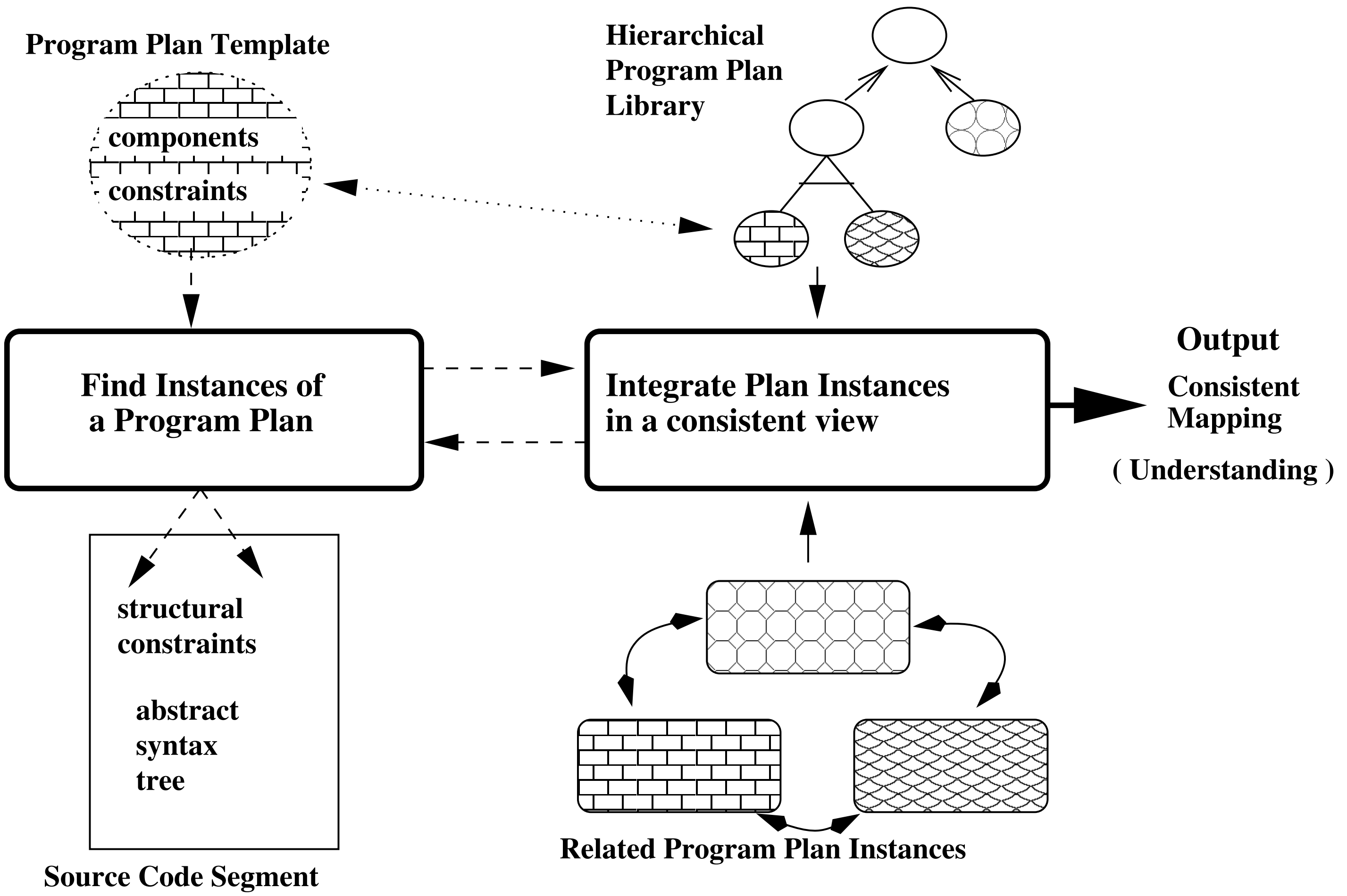
```
main()
{
  // documentation describing an older version of code
  char* A, B, C;
 A = "s" + "t" + "r" + "i" + "n" + "g" + "1";
 B = "string2"
 // description of non-existing variables, omitting current
 sz = 7;
 for (int j = sz;  j>0; j--){
     C[sz-j] = B[sz-j]; }
 C[sz] = "3";
 for (int i=0; B[i]; i++){
      print(B[i]); }
 // comment referring to non-existent code
 for (int j=0; j++){
      printf("%s",C[i]); }
 // reference to outdated documentation, old library
 for (int k=0; A[k]; k++){
     outchar(A[k]); }
 }      // end of main
```



Problem: The Basis of Program Understanding

- What is the **knowledge** we map from ?
  - → Existing program plans
  - → Plans encoded in Abstract Data Types: data + function
  - → Plans encoded in Class, Template software libraries
- What are the **artifacts** we map to ?
  - → Code can have functional segment structure
  - → Parsed source yields Abstract Syntax Trees
  - → Extracted Control and Data-flow constraints

Our View: Two-Phase Program Understanding



Program Understanding: What's the Problem ?

if Polynomial algorithm exists

then Design one to directly solve the problem

else Justified in using heuristic methods for problem

  but

    How do we *generate* these methods ?

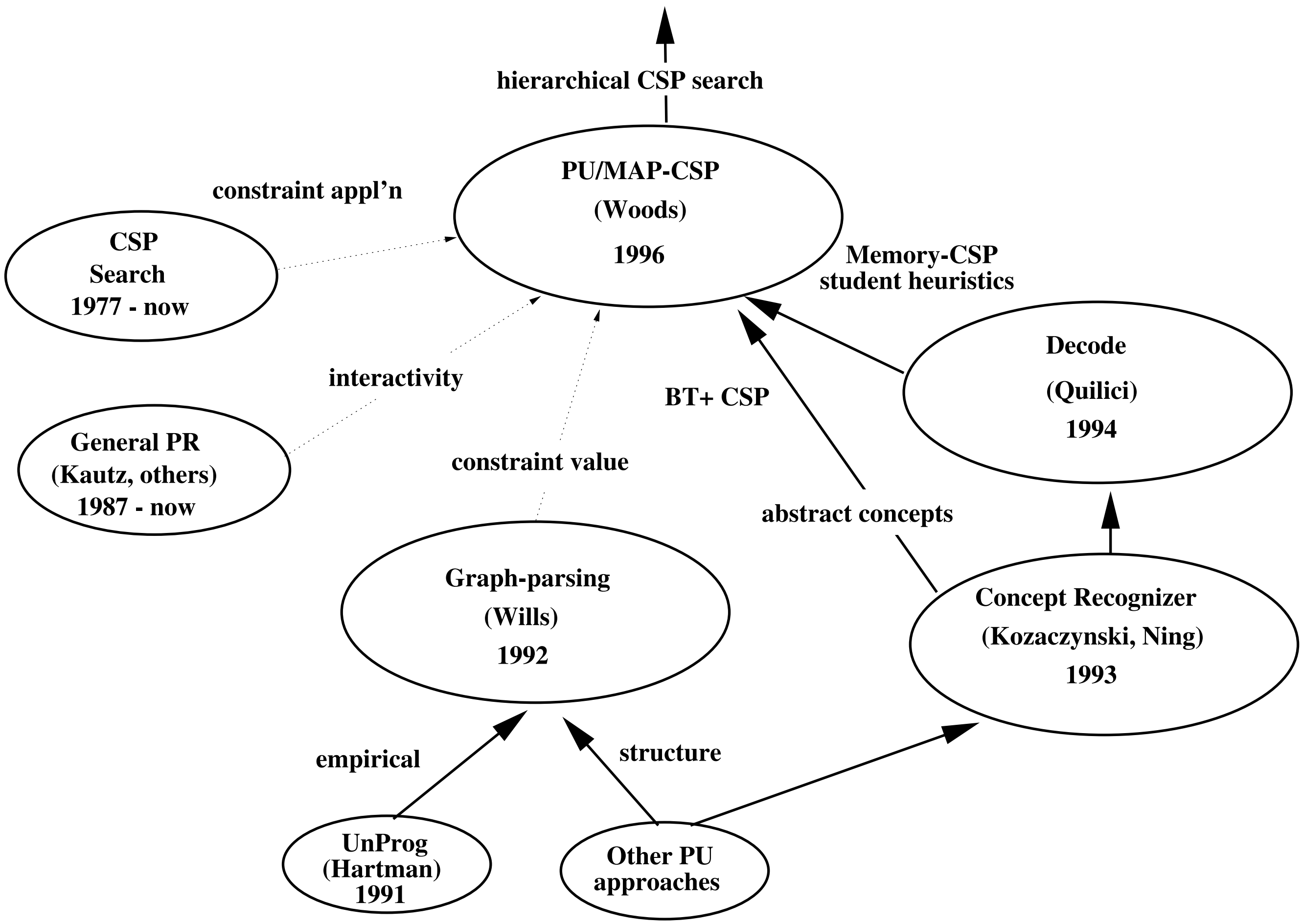
    How do we *select* among these methods ?

    Can a heuristic method *scale* for useful cases ?

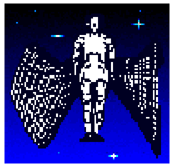
Motivation: Program Understanding

- Past work assumed Program Understanding non-polynomial
- Existing approaches have a series of drawbacks:
  - → Not known/shown to scale with program size
  - → Hard to compare, combine rep'ns, search strategies, heuristics
  - → De-coupled representation, search
  - → Fail to focus on search explicitly as scaling factor
  - → Fail to exploit hierarchy, constraints, interation well

Recent Program Understanding World



Direction and Sufficiency of Research



Step 1 Clearly establish Program Understanding complexity

Step 2 Construct search-based model of Program Understanding

→ Implement CSP partial-understanding model

→ Extend CSP algorithms : hierarchy

Step 3 Establish initial empirical benchmarks of CSP-model(s)

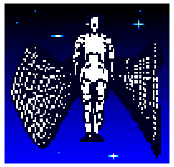
Step 4 Integrative, Interactive Method of Understanding

Step 5 Adapt previous strategy(ies) as CSP as examples

→ Implement previous strategy(ies) as CSP

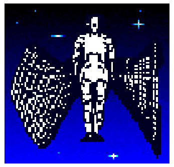
→ Contrast CSP-models: empirical, analytical

$Step_1$: Complexity Established NP-hard

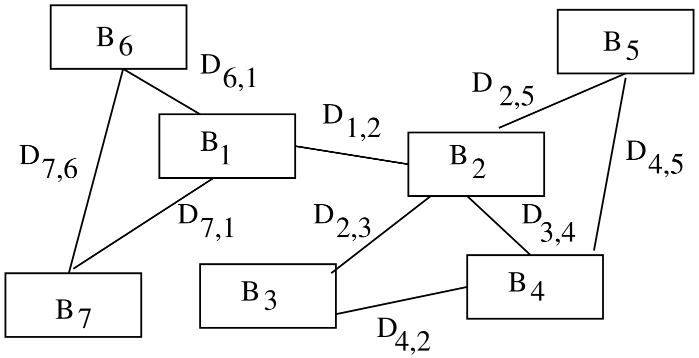


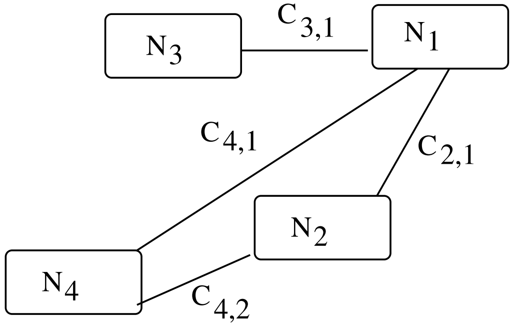
- SPU: Global (Integrative) Program Understanding, NP-hard.
  - Reduction from Subgraph-Isomorphism
  - SPU simplifies General Understanding, PU-CSP
  - Find program *structural* graph in library *knowledge* graph
- SMAP: Partial (Local) Program Understanding, NP-hard.
  - SMAP simplifies Partial Understanding, MAP-CSP
  - Find given *knowledge* sub-graph in program *structural* graph
- Motivated to find better heuristic solutions via search.

$Step_{2a}$: MAP-CSP: Local, Partial

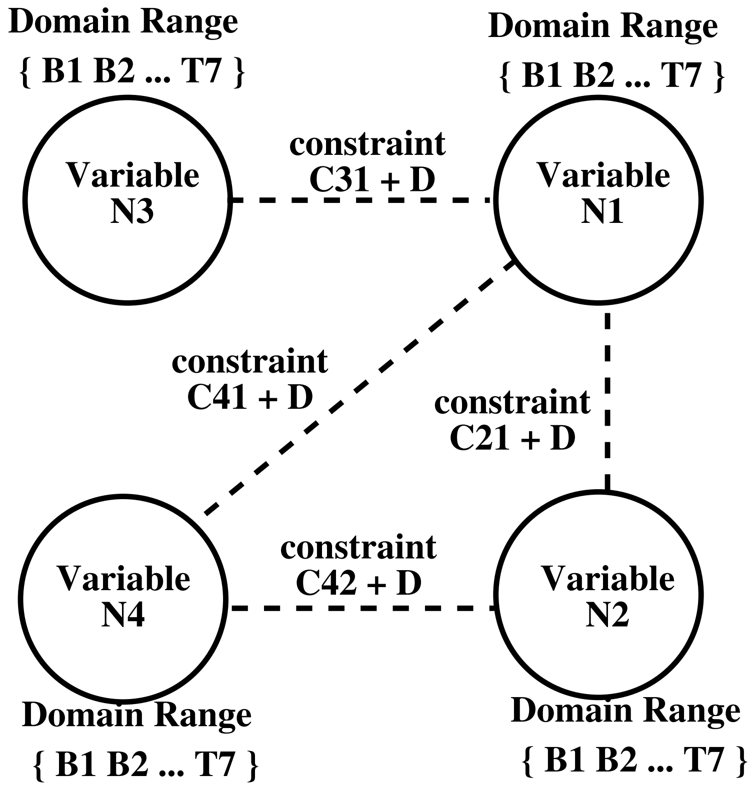


Source Program

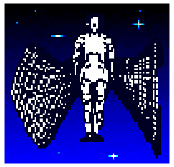


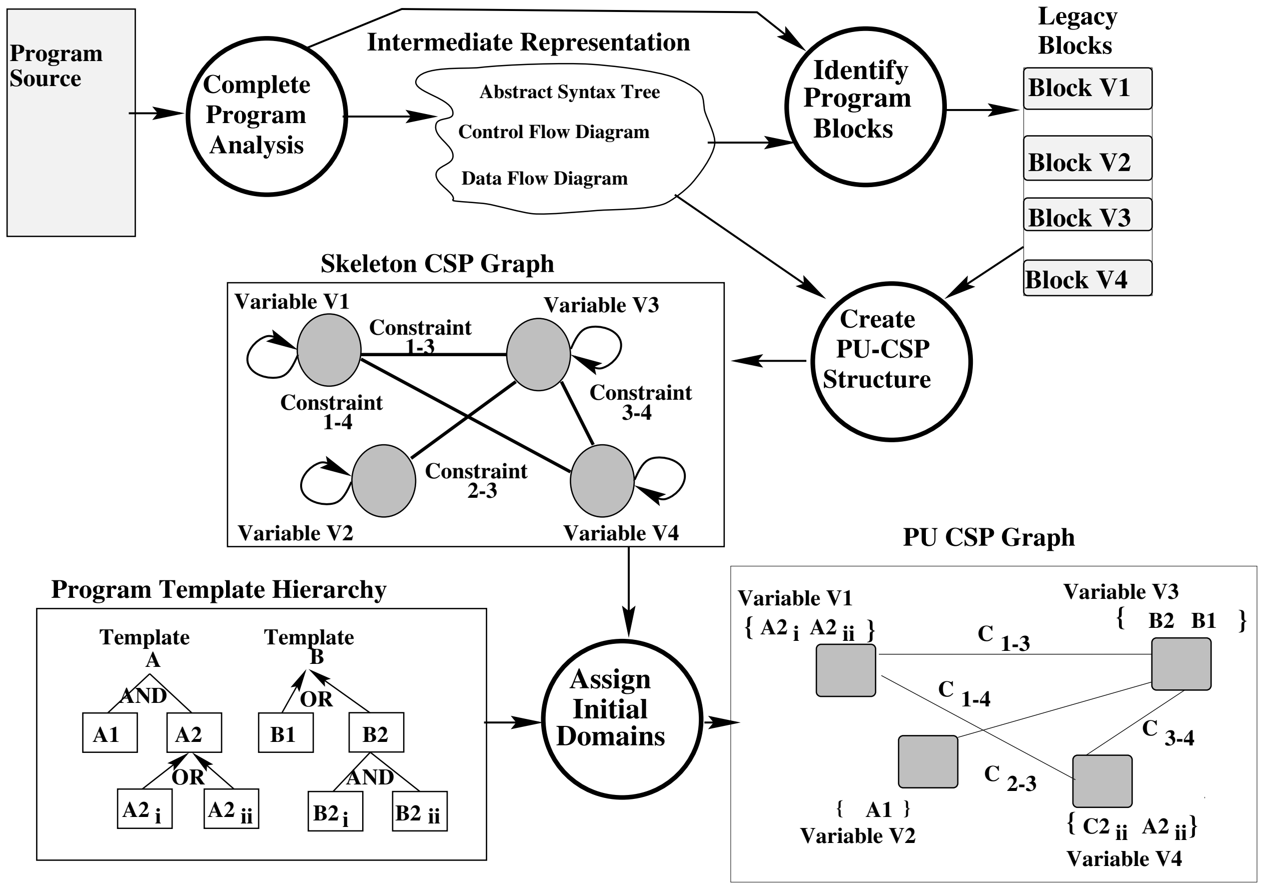


Program Template Plan to Match

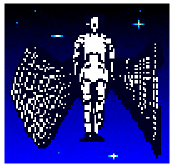


$Step_{2b}$: PU-CSP Global View

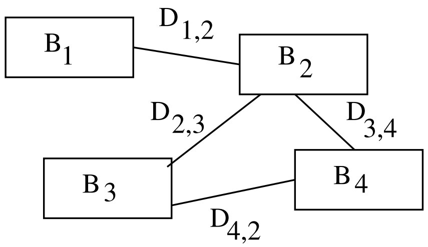


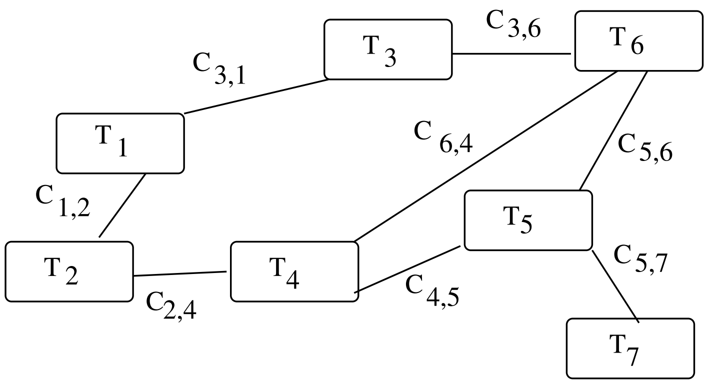


$Step_{2c}$: PU-CSP: Global, Integrative

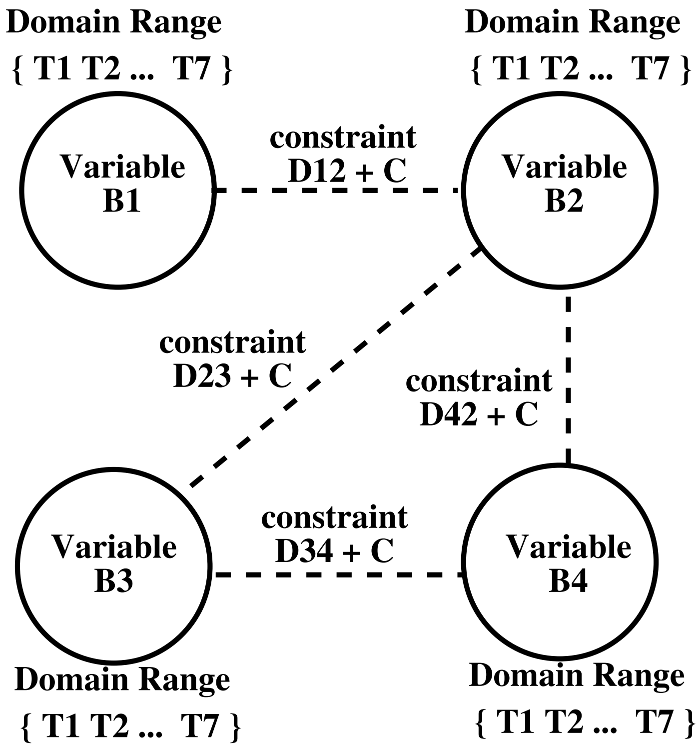


Program to Understand

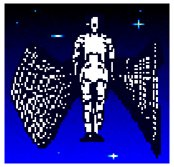




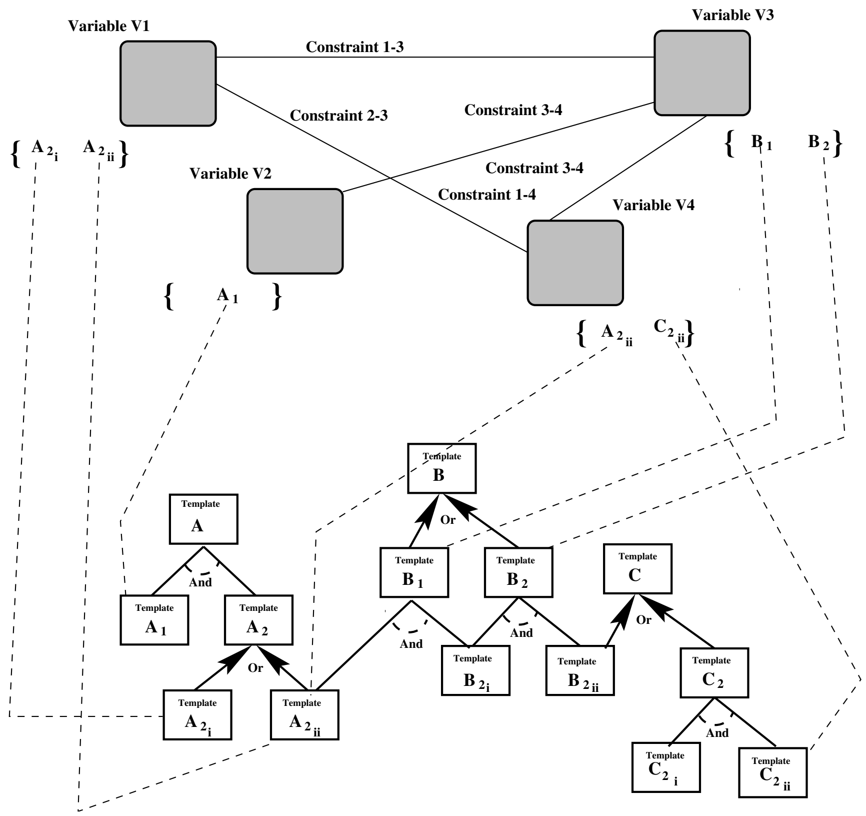
Library of Program Template Plans



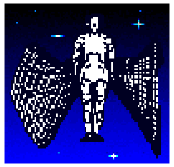
$Step_{2d}$: Hierarchical CSP Extension



PU-CSP Graph (node consistent)

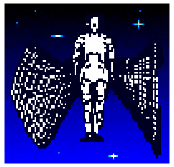


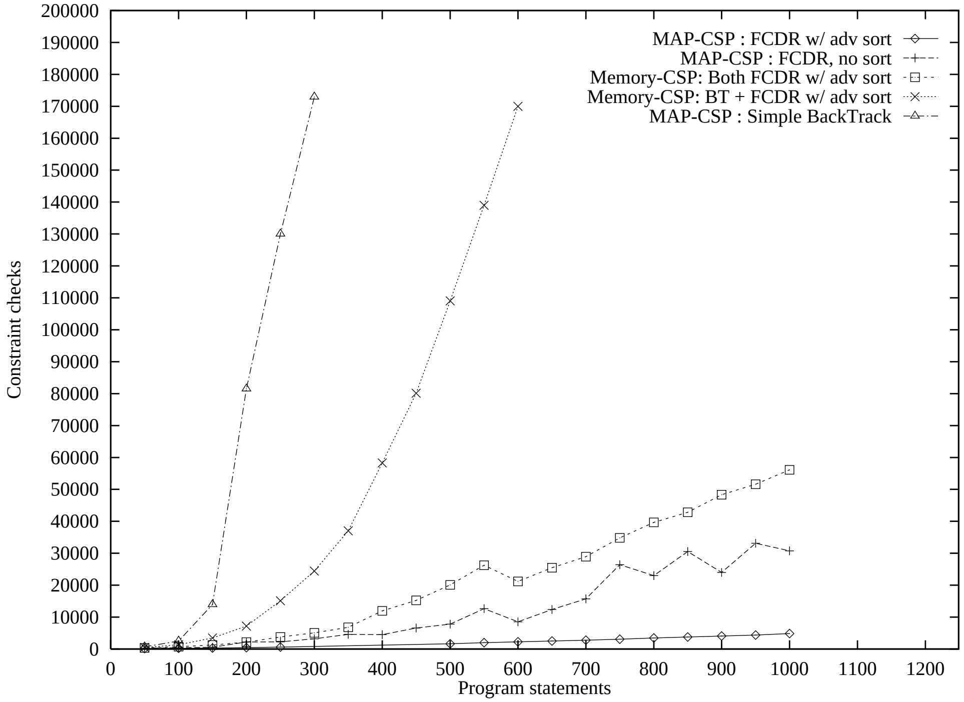
$Step_3$: Solving CSPs



- CSP Search
  - $\rightarrow$ Intelligent Backtracking - FC/DR, BT, BJ, BM
  - $\rightarrow$ Dom Indep Heuristic: variable/constraint ordering
  - $\rightarrow$ Dom Dep Heuristic: Memory-CSP/indexing
- CSP Constraint Propagation
  - $\rightarrow$ Arc consistency
  - $\rightarrow$ Partial Arc consistency
- CSP Hybrid Search (future experimentation)

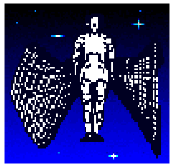
$Step_{3a}$: MAP-CSP Empirical Results

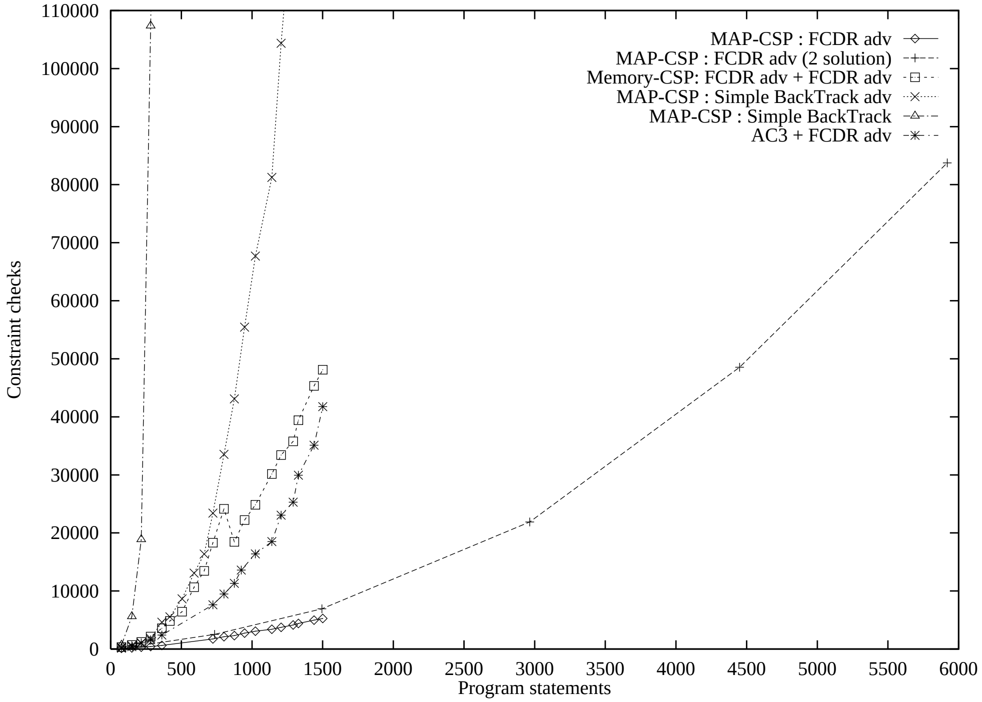




- Matching a Template with 9 components, 20 constraints.
- MAP-CSP approach *dynamic* for given problem instance.

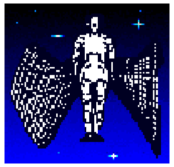
$Step_{3b}$: MAP-CSP Empirical Extension





- Lisp Implementation, 1500 LOC completes in $<$ 35 seconds.
- Memory-CSP 1500 LOC $\rightarrow$ 10x work; Same work $\rightarrow$ 3x LOC.

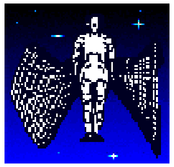
$Step_{3c}$: Experimental Summary



- Demonstrated scaling to **larger ranges** than prev shown
- New Source-size **thresholds** for search comparison
- Consistent results with larger templates, source dist'ns
- Instance/Domain-independent strategies shown effective
- Shown explicit advantage to constrainedness in search

$\Rightarrow$ Additional constraints will boost performance

$Step_4$: Integrate Hierarchical PU & MAP



D, domains; B, blocks; L, library; S, source; T, template;

1 $D := InitializeDomains(B, L)$; */* Input/Output Matching */*

2 $D :=$ AO-HAC (OR NEW)(D, L); */* Hierarchical Arc-Consis */*

3 $T := SelectPlanTemplate(D, L, S)$; */* Local Plan Inquiry */*

4 $Tinstance_{set} := MapCSP(M, S)$; */* Partial, Local Expl */*

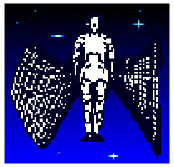
5 **Loop ForAll** T_i in $Tinstance_{set}$ */* Propagate Local */*

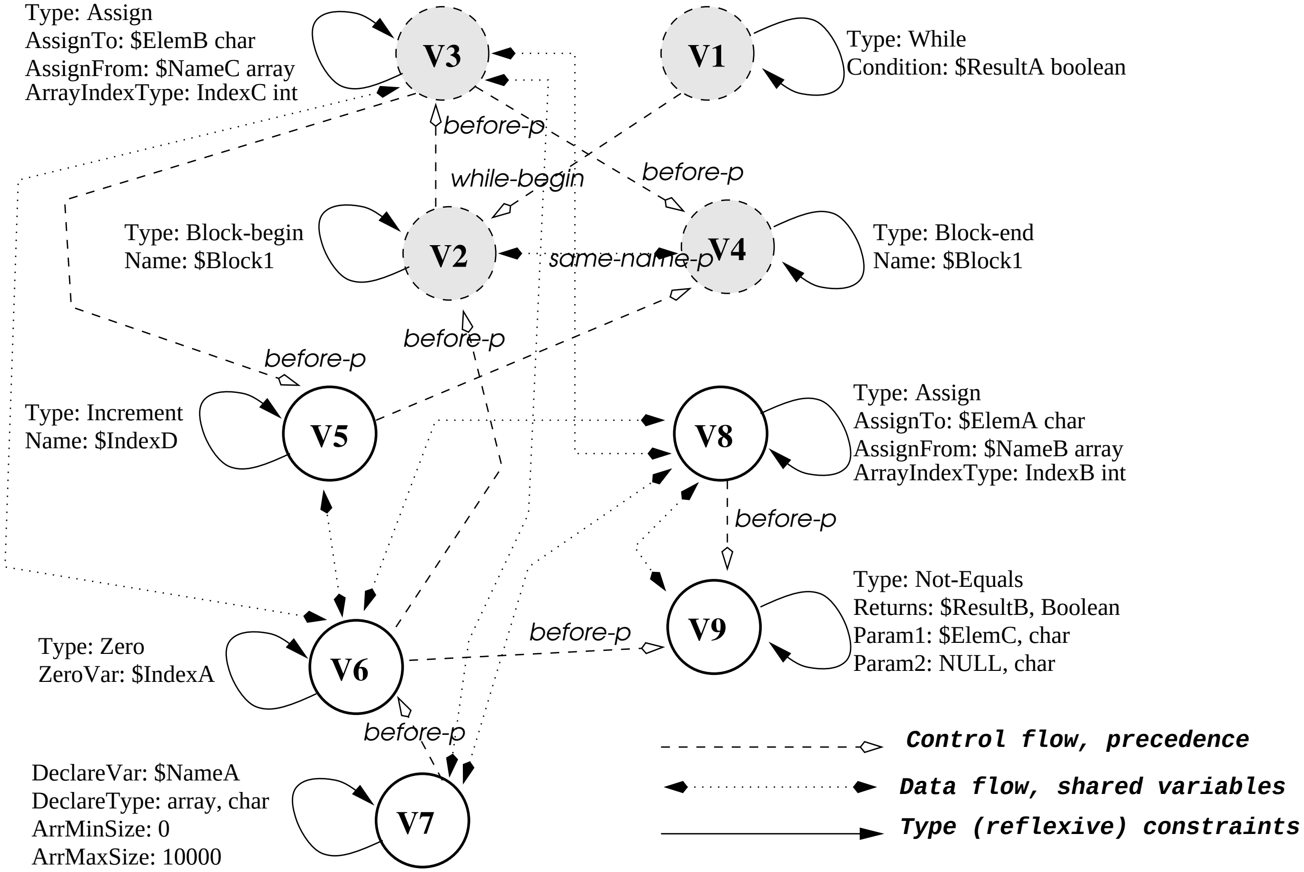
6 $\quad D := MergeRevise(T_i, L, S, B, D)$;

7 Optionally, MergeRevise negative information about T matches

8 **if** (**not** $Done$) **then GoTo** 2 **else Exit**;

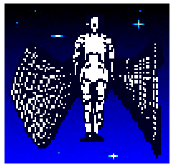
$Step_5$: Memory-CSP: DECODE





Partition CSP: Identify Index Instances; Resolution

Ongoing & Future Research



with QUILICI : Extending constraint structure: parsed AST+df/cf

w. QUILICI : Integrate MAP-module in rev-eng toolset

w. QUILICI,YANG : Extend Plan Recognition Comparisons

w. YANG : Appl'n of MAP-module to Y2K-type probs

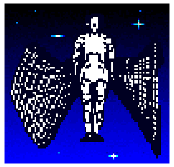
w. KAZMAN : Appl'n MAP-module to architectural re-engineering

w. DONALDSON : MAP-module Local-search Experiments

w. DEVANBU,SELMAN,KAUTZ : MAP-module GSAT Experiments

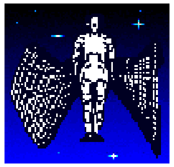
w. QUILICI : Extension of comparative impls to include Wills

Conclusion: Contribution Summary



- Simplified Program Understanding (SMAP, SPU) NP-hard
- Two-Phase CSP Model (MAP/PU-CSP) for Understanding
  - → MAP-CSP model prototyped; Empirical comparisons generated
  - → Improved Scalability-potential demonstrated
  - → Adapted 2 earlier strategies as CSP; Empirical comparison
- Hierarchical CSP extension proposed, implemented
- PU-CSP Interactive Standardization Scheme (Select, Integrate)
- Program Understanding special case of Plan Recognition

Thesis Publications



- Jrnl - CSP Modelling. *Automated Software Engineering*, 1996.
- Jrn/mod - MAP-CSP Empirical. *Automated Software Engineering*.
- Conf. - Empirical. *Work. Conf. Rev. Eng.*, Sept., 1996.
- Conf. - Plan Recognition. *Know. Based Soft. Eng.*, Sept., 1996.
- Conf. - NP-hardness. *Int'l Conf. Soft. Eng.*, March, 1996.
- W'shop - CSP Modelling. *W'shop Prog. Compr.*, March, 1996.
- *Conf. - MAP-CSP, empir. *Work. Conf. Rev. Eng.*, 1995.
- *W'shop - MAP-CSP, empir. *Int'l W'shop Cmpt. Aid. Soft. Eng.*, 1995.
- W'shop - PU as CSP Model. *IJCAI W'shop AI & Soft. Eng.*, 1995.